METHODS & TOOLS

Practical knowledge for the software developer, tester and project manager **ISSN 1661-402X**

Spring 2009 (Volume 17 - number 1) www.methodsandtools.com

Between the Magic Quadrant and the Bermuda Triangle

Software vendors are always proud to communicate evaluations from research companies that indicate how good their products are. These ratings are to the software development tools market what the gastronomic guides are for restaurants. A good position in these evaluations is something sought after, especially by smaller and young companies, as it could be a driver for their future growth. Even if these external assessments are still important for the software development market, the rules have certainly changed these recent years. The relationships between software tools vendors and developers have been transformed, both in their form and in their content.

The investments needed to launch a new software tool have decreased as a consequence of the diminution of the solid content of software tools. You are no more obliged to build a physical distribution network and to manufacture software CDs and documentation manuals. Everything can be available on-line and the vendor sells directly to the buyer, everywhere on the world. In 1993 a manager told me that the barriers for growth were high, because for each dollar you spend to develop a product, you needed ten dollars to sell it. The relative marketing/product cost ratio has notably decreased, moreover if you target small and mid-size customers. A second major change in the software market rules is that some vendors are not selling software anymore, but only the services provided by the software. Amazon, Google and other Web-based tools vendors are not selling a database software, but the service of using a database to store and retrieve your data, located somewhere on the net. The advantage of this solution is that you don't have to install and manage your tools and your costs are directly related to your needs and activity. You become also more dependent on your supplier and this could not be without risks. A third major difference is the growth of software-less software companies. Companies like Red Hat or Spring Source do not sell software tools. They sell mainly the support and the consulting that goes with open source tools. What will be the impact of the current economic crisis on the software tools market? Do these new market rules change something? I think that the new "nimble" vendors and their low-cost Internet models could have more chances to survive longer. The "no license cost" aspect of open source software could event lure people to switch to open source solutions.

This issue has more than to 80 pages. Is this too long in the age of Twitter? I don't think so. As companies could cut training cost, Methods & Tools remains committed to supply software developers worldwide with quality articles. Do not hesitate to share this free content with your friends and colleagues. A final word to thank our longtime sponsors: MKS, TechnoSolutions and e-Valid. They are keeping their support to Methods & Tools in this difficult period for the economy.

[illegible]

Inside

How to Build Articulate Class Models and get Real Benefits from UML

Leon Starr, leon_starr @ modelint.com
Model Integration, LLC, www.modelint.com

Abstract

The typical UML class model is a nebulous representation of the reality it aspires to formalize. This, at least, has been my experience as a longtime executable UML modeler and project consultant. What I am defining as an “articulate” class model is one that expresses critical system rules with transparent, unambiguous precision. The contrast is best demonstrated with a good vs. bad model example. This won’t be a contrived model comparison, but one representative of the sort of thing seen all the time on real projects. I will also itemize the negative consequences of an imprecise class model on a software system. And, of course, I will point out the practical benefits of doing things the right way. Finally, I will describe some simple techniques you can use to create more articulate, rule-expressive class models.

A UML class model should do more than just give you a pretty picture of your data structures. The official term is “class diagram”, but it is often useful to distinguish between a model which has no particular visual representation and a diagram which implies a 2D visual representation of the model with a particular notation. Both a statechart and a state table may be used to visualize the same underlying state model, for example. I only use the term “diagram” when I am referring to the graphical notation. An articulate class model will nail down subtle, yet critical, constraints in your application. It will expose hidden rules and assumptions that, when overlooked, lead to the nastiest of bugs down the road. A good model will head off key design miscalculations and save you and your team a lot of time and pain. If you aren’t getting these kinds of results from your class models, you probably shouldn’t bother with them at all.

Common UML Class models, both on projects and in industry literature, barely scratch the surface of their true potential. Key application constraints are often ignored. Models are open to multiple interpretations. This is often justified under the lazy mantra of “Sort it out in design”. Well, if all the hard problems are going to be sorted out in design, why not just start designing now?

The Class Model is Misunderstood

This trouble starts with a key misunderstanding. Class models are often described in the UML literature as “representations of static structure”. In other words, it is understood and oft repeated that a class model does not reflect dynamic behavior. The fun stuff, after all, is in the statecharts, sequence diagrams, action language, collaborations and activity diagrams. And since the dynamic properties of your software constitute its observable features, it is easy to conclude that class models are the obligatory vegetable dish in the all-you-can-eat meat buffet of the UML. You are supposed to model the classes and relationships first, you are not quite sure why, and you want to get it over with as soon as possible.

Static Model Can Express Dynamic Rules

Yes, class models *are* static. Then again, so are *rules*. Consider a rule like “Only one aircraft may take off from a runway at a time.” If all goes well, this rule shouldn’t be changing during runtime. An air traffic control system comprises thousands of such rules, some intuitive and obvious, others devilishly subtle. The behavior, intelligence, resilience, and reliability, in short, the dynamic personality of your software is defined and constrained by an application rule base.

Every application has one and it is nice to crystallize the rule base in a platform independent manner. Few developers appreciate the degree to which class diagram notation can mold and constrain complex system behavior. But to get this result, you have to stop thinking about class models as mere repositories of data.

So what is Wrong with an Inarticulate Model?

If the rules aren’t expressed in your class model, they have to go somewhere, don’t they? Any rules not captured in your class model are deferred to the statecharts. If not captured there, the rules are deferred to the state actions. If you are writing the code by hand instead of using a model compiler, you have yet another opportunity to catch the neglected rules in your code. Beyond that, let the users find them for you. If you are building avionics, anti-lock brakes or medical systems, that last option is considered bad form. I just downloaded the latest beta into my iPacemakers! It's so kewl... wait a minute, that's odd...

The Benefits of Articulate, Rule-Expressive Class Models

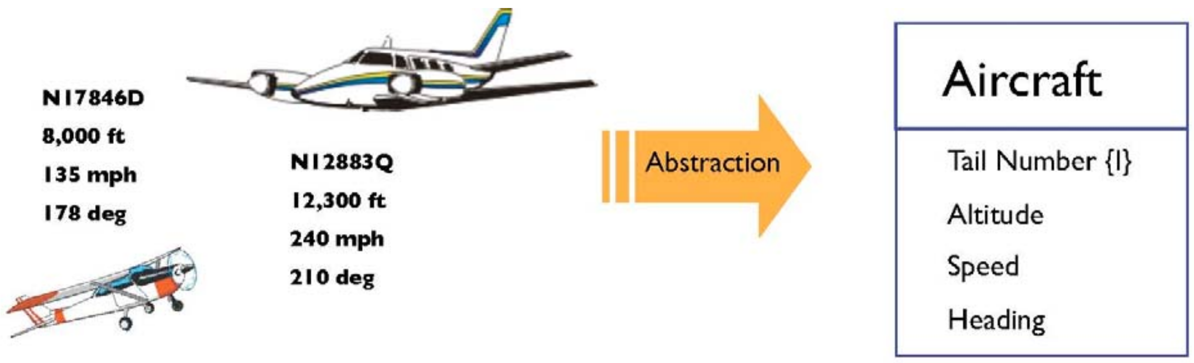
Why put rules and constraints in your class models? 1) Many rules are easily and more efficiently expressed in the class model once you know how. 2) Rules in a class model are highly visible to users and application experts who can provide critical guidance before the design solidifies. You can step an expert through an articulate class model and get quality input faster than having them absorb pages of sequence diagrams. 3) Increased visibility tends to force key application decisions, which are easily swept under the table until they emerge, when it is too late or expensive to make the necessary changes. 4) States and actions involve sequence, synchronization and timing which is generally harder to test and prove correct than “timeless” static structure. 5) Fewer rules in the actions means less code, more data perhaps, but it is easier to optimize a design to handle data e.g. choosing storage and access for read-only specifications vs. runtime values. 6) Rules expressed in data can be tuned and updated without having to change, recompile and retest the code. 7) By focusing on the data early on, you are more likely to identify and abstract configurable parameters than hard wire them into a procedure. With rules modeled in data you get the best of both worlds - configurability and hard enforcement of core principles. 8) The precision questions that must be asked and answered to build an articulate model will make you smarter - or at least create the appearance. (I have seen this last feature turn newbies into experts so fast that the newbies end up taking control of the project from the resident software cowboys who assume that no-one is smart enough to learn their stuff). Articulate class modeling is not just a software development technology, it’s a brain development technology!

I could keep going, but let’s get to those examples!

Capturing multiple rules with a single class

Let’s start out simple. We’re going to lay out a lot of groundwork with just one single class. So bear with me while we delve into the nuts and bolts. Once we get through this, we can start scaling up to something more interesting. You may be surprised, though, at just how many rules can be expressed in a single modeled class.

Say that we are tracking multiple aircraft flying around an airport. We want to avoid collisions and know where all the aircraft under our control happen to be at any given moment. So our software must maintain an internal representation of each aircraft instance flying around.



Abstracting a class from data in the real world

We've abstracted a class called "Aircraft". But what does this abstraction mean exactly? Is it a Java class? Is it an Objective-C class? A Python class? Is it a database table? Is it a section of an XML file? Much of my work is in embedded systems so the answer is often some tight C or Assembler structure. Each implementation technology carries a different set of assumptions and limitations. But our abstraction is none of the above, it's a UML class! But what does *that* mean? This is THE big problem with UML and modeled abstractions in general.

Advertisement - Visual Use Case - Click on ad to reach advertiser web site

Crystallize your Requirements

Document Requirements precisely with "Structured Pseudo-Code"

...and then with a single click...

...convert it into a "Flow Chart" that everyone can understand!

Visual Use Case™ 2006

Write better Use Cases in less time. Multiply Productivity, Improve Software.

Click here to download your instant free trial of Visual Use Case.
http://www.TechnoSolutions.com

By contrast, when you look at good hard code it is easy to roll the mechanics of a problem around in your head and visualize “what happens if”, “what can’t happen”, and so forth. A serious modeling language requires more than just graphical notation. There must be unambiguous semantics (meaning) underneath the notation so that, like code, a model means one thing and only works one way. It shouldn’t be open to wishful interpretation.

Defining a Platform Independent Class

Unlike code, we would like a platform independent specification (as much as possible) so that we can separate the real world (application) rules from the rules introduced by a particular implementation. What we care about are the aircraft management rules that must be enforced in *every* implementation. Rules about minimum vertical separation distance between two flying aircraft don’t change just because the tracking system happens to be running on Windows Vista (Insert your own joke here).

Even though we are using UML, we must think about a class as an actual data structure. A simple rectangle on a sheet of paper has no interesting mechanics. A whole bunch of rectangles on a sheet of paper is a colossal waste of time. But once we agree on an UML class data structure we can evaluate the operations that can be applied to the data in that structure. That’s when real thinking can start. But we need a platform independent data structure - is this possible? In fact, thanks to mathematics, set theory, and the relational model of data, there is. A platform neutral data structure for a class is simply a relation - less formally, a table. Not a relational database table (that would be one possible implementation).

Here is what the Aircraft class might look like as a platform independent table:

Aircraft

Tail Number {I}	Altitude	Speed	Heading
N17846D	8,000 ft	135 mph	178 deg
N12883Q	12,300 ft	240 mph	210 deg

A populated Aircraft table

The {I} on the Tail Number attribute is a UML tag used as shorthand for the identity constraint [1]. It signifies that there can be no two duplicate values of Tail Number in the table. So at this point we see that the rectangle class symbol in our abstraction can be understood as a tangible data structure. Operations for manipulating data in a table are well defined in relational algebra and supportable by UML actions. Those of you familiar with relational theory will see that a class is represented as a relation in third normal form (3NF). There is a nice Wikipedia entry on the topic. We will use some of these actions to access the Aircraft table data shortly.

Just to be clear, we are not specifying an implementation. The table above could be realized by a variety of implementation structures. The choice will depend on the target platform, read vs. write access optimization, and other performance and programming language characteristics. Our goal is to specify as many platform invariant application rules as possible without demanding anything in particular of the implementation. So the programmer (or model compiler) is free to package this stuff up as he, she or it sees fit as long as all application rules are perfectly preserved. In practice, I have seen tables transformed into everything from C++/Java classes to arrays and lists of structures in C and even simple bitmapped fields in assembly language. A few people are tinkering with translation directly to Verilog (hardware).

Expressing Application rules with an Empty Table (Platform Independent Class)

Now that we have our platform independent class data structure, we can get back to expressing what's really important - our aircraft rules! Since an application's rules are the same regardless of what instances happen to exist at a given slice of time, let's remove the data and focus on the structure itself, an empty table.

Aircraft

Tail Number {I}	Altitude	Speed	Heading

Empty table is a platform independent data structure

The distinction between structure and content (population) is key. While we often examine the data (instance populations) to abstract general principles and rules, it is the abstraction (structure) that we use for generating code. We throw the data away once we derive the abstractions. But on the way to our abstractions, especially in a complex system, we may plow through a LOT of data searching for patterns and subtle distinctions among instances.

Advertisement - MKS Integrity - Click on ad to reach advertiser web site

REALIZE THE VISION FOR UNIFIED ALM
MKS INTEGRITY™
Go Agile with Unsurpassed Collaboration and Flexible Process
Envision a solution with the flexibility to support both Agile and traditional processes.
Envision a team able to collaborate in real-time across the lifecycle.
Envision having full visibility into projects with burn-down charts and reports based on live data.
Turn Vision into Reality. Go Agile with MKS Integrity for Unified ALM.
www.mks.com/solutions/agile | 1.800.613.7535
MKS

Without worrying about what data happens to be in the table, let’s take an inventory of rules expressed by this simple structure.

Aircraft class rules

1) Each aircraft is identified by a unique tail number.
2) No two aircraft may have the same tail number.
3) Every aircraft has an altitude.
4) Every aircraft has a heading.
5) Every aircraft has some airspeed.

Rules 1 and 2 are established by the identity constraint described earlier. We’ll talk more about how the constraint is actually enforced in an implementation later. For now all we know is that it is unambiguously declared on the class model.

Rule 3-5 are consequences of a relational rule which states that every cell must contain a non-null value. It’s a nice rule that leads to sharper, less wishy-washy class abstractions. In this case, you have to ensure that all flying things covered by the Aircraft definition do in fact always have legitimate values per attribute at all times.

What Questions Can This Class Answer?

The power of an underlying data structure is best demonstrated by asking the model some questions and seeing if it can answer them. For each question, ask yourself if it can be answered by the populated table.

1) What is the heading of N12883Q?
2) Which aircraft are below 10,000 ft?
3) How many aircraft are in our Control Zone?
4) How fast is N17846D going?
5) Is a collision likely in 5 minutes?

Question assessment

(1) Yes. We can scan the Tail Number column, pick out “N12883Q” and following the row to the value in the Heading column. If we don’t find the Tail Number, we conclude that there is no such Aircraft in our area.

(2) Yes. We scan down the Altitude column selecting each row with a value < 10,000 ft and then jump across the row and report the Tail Number.

(3) No. The model does not say anything about a Control Zone. (And neither did I until I listed the question!) You might expect to find another class titled Control Zone and some attribute reference in the Aircraft table to a Control Zone ID of some sort. But there is none in this example, so the answer is “No”.

(4) Yes. Find the Tail Number as in (1) and scan across to the Speed column to get the value.

(5) No! We have insufficient information to compute collision possibilities to any useful degree of certainty. That’s because we don’t have enough spatial coordinates for each Aircraft. We

know where an Aircraft is pointing, how fast it is going, but we don't really know where it is. Easily fixed, though:

Aircraft

Tail Number {I}	Altitude	Speed	Heading	Lat	Long

A smarter table that can answer more questions

Now we can answer question (5).

Notice that we really didn't need the populated table to evaluate our questions. We could use the empty Aircraft table and just rephrase the questions as follows:

1) What is the heading of a specified aircraft?
2) Which aircraft are below a specified altitude?
3) How fast is a specified aircraft going?
4) Is a collision likely in a specified time period?

All of these questions can be answered by the improved empty table, though you might insist on adding "Climb Rate". It's an interesting piece of data since it is computed from what we already have. Consequently, we can argue that the table is smart enough to answer the collision question with or without this derived attribute. It's reasonable to add it though, so let's go back to our UML Class notation and put it in.

Aircraft
Tail Number {I} Altitude Speed Heading Latitude Longitude /Climb Rate

Tail Number {I}	Altitude	Speed	Heading	Lat	Long	/Climb Rate

The / in front of Climb Rate indicates that the value is derived computationally. The judgement of whether or not to add computationally derived attributes is the subject of a whole other article, so I'm going to just say "Put them in as needed." for now.

UML and Underlying Semantics

To recap, we've looked at a trivial example, a single class, and observed that several rules are already captured. The table representation gives us a hard data structure so that we can think about the mechanics of data access without regard to any particular implementation. In the process we found that we could make definitive statements about what we know, what we can compute and what we cannot compute. This all comes without knowing *anything* about the statecharts, actions or algorithms. Just the drawing of a single class puts us in a position to answer questions and to evaluate conformance with the world we plan to control. Just imagine how many rules can be captured when we add relationships and hundreds of classes! There is a direct and fascinating correspondence between relational elements and logic theory described in [4].The gist of it is that a relational model is essentially a set of logical predicates.

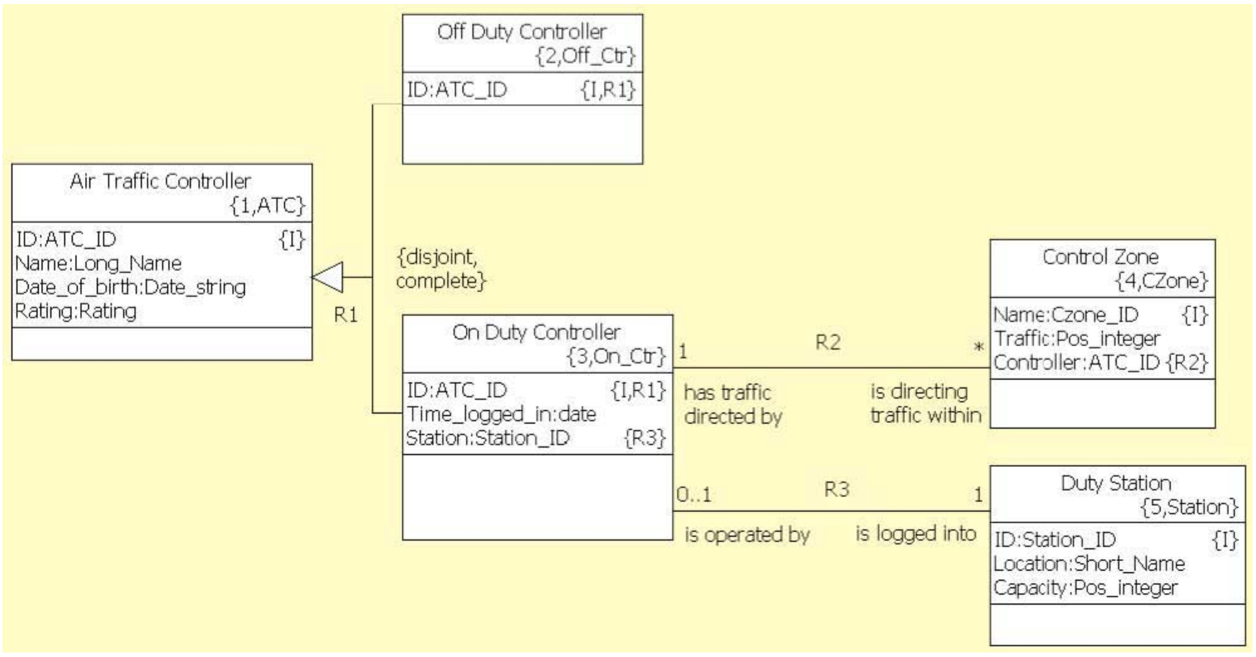
Enforcing the Identity Constraint

You might wonder where the {I} constraint is actually enforced. The class model simply states the identity requirement. When code is generated, by a programmer or model compiler, there should be a systematic mechanism for handling identity. An embedded architecture might make use of a hash table whereas an enterprise architecture might rely on available database mechanisms for detecting and rejecting duplicate entries.

A Good Model

Moving along to our model comparison, we start with the "good" model then see what's missing in a "bad" model example. I apologize for any narcissism in presenting *my* models as "Good Models" and everyone else's as "Bad Models". As a default tendency it's not a constructive (and is an often wrong) practice! Naturally, we need to focus on tangible practical differences that can help us all improve analysis and modeling technique and generate better code.

Here is the class diagram of the purportedly Good Model.



A Good Model

Some of the notation above is familiar UML, but there are a few twists. Let’s just go class by class and see what’s going on underneath all this notation. We’ll zoom in to the underlying table data structure and then pull back and evaluate big picture again. After that, we will finally be in a position to contrast this model with a Bad Model.

Air Traffic Controller Classes

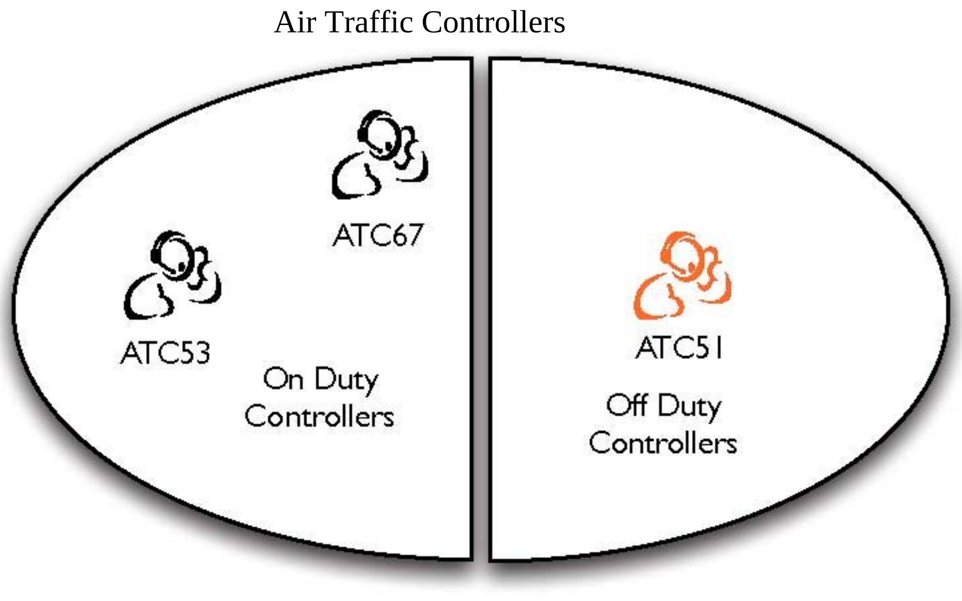
The left half of the model features three classes: Air Traffic Controller, Off Duty and On Duty Controller. The superclass table, populated with some sample data, might look like this:

Air Traffic Controller

ID {I}	Name	Date_of_birth	Rating
ATC53	Toshiko	Jun 12, 1975	A
ATC67	Gwen	Mar 28, 1981	B
ATC51	Owen	Dec 23, 1974	C

We have three Air Traffic Controllers and we know the name, age and overall skill (rating) of each. We also see that ID values must be unique since the {I} tag is present. This table contains data relevant to an ATC regardless of on or off duty status.

The {disjoint, complete} text next to the generalization relationship R1 is a pair of standard UML tags. It indicates that every Air Traffic Controller object is either an On Duty or an Off Duty object. The “disjoint” part means that it is not possible to be both on and off duty at the same time. The “complete” part means that every Air Traffic Controller is definitely on or off duty. Work status is defined at all times for each ATC. And it is certainly not possible for an On or Off Duty object to not be an Air Traffic Controller object! So R1 does not represent inheritance style generalization. It is more akin to relational set subtyping. It is the XOR of class modeling - an incredibly useful tool for incorporating logic into data structure.



Disjoint - Complete Generalization Relationship

On Duty Controller

ID {I, R1}	Time_logged_in	Station {R3}
ATC53	9/27/08 3pm	S2
ATC67	9/27/08 11am	S1

When an ATC is on duty, he or she must be logged into a Duty Station. Both the Station ID and the login time is kept for each of these ATC's. This data is relevant only while on duty.

Note the {R3} tag on the Station referential attribute. This tells us that the "Station" corresponds to the identifier on the other side of R3. Duty_Station.ID, in this case. This is another example of platform independent data structure mechanics. You can ask the question: "what station is ATC53 logged into?" It is clear from the table structure that you can just find the row containing ATC53 in the ID column and scan across to the Station column and return the value,"S2". You can go even further and retrieve data from the Station table using S2 as a key. So you can answer a more complex question like "What is the location of the Station logged into by ATC53?". You can query in both directions on any class model relationship, so you can also answer the question: "who is logged into Station S2?" Just scan through the On Duty Controller table in the Station column, find the row and return the ID value, ATC53.

The point of all this row/column scanning is simply to show that the data is connected somehow in any implementation. It doesn't mean that we need to implement our actions in SQL or anything like that. The systems I work with usually strip out the referential attributes and replace them with pointers, handles or indices of some sort. Access directions on each association are typically optimized or disabled as necessary. Some model compilers will, for example, scan the action language, see that no write accesses are made in a particular association direction and omit the code for the relevant accessor. Model compilers exist which will do this automatically yielding nice C, Java or C++ and, believe it or not, even Assembler! Generating some kind of SQL is certainly an option for database-y types of systems, though.

By the way, note that there is an R1 tag on the ID attribute indicating that it is simultaneously an identifying attribute of On Duty Controller which matches its ATC superclass ID. All relationships in the Good Model are glued together with referential attributes.

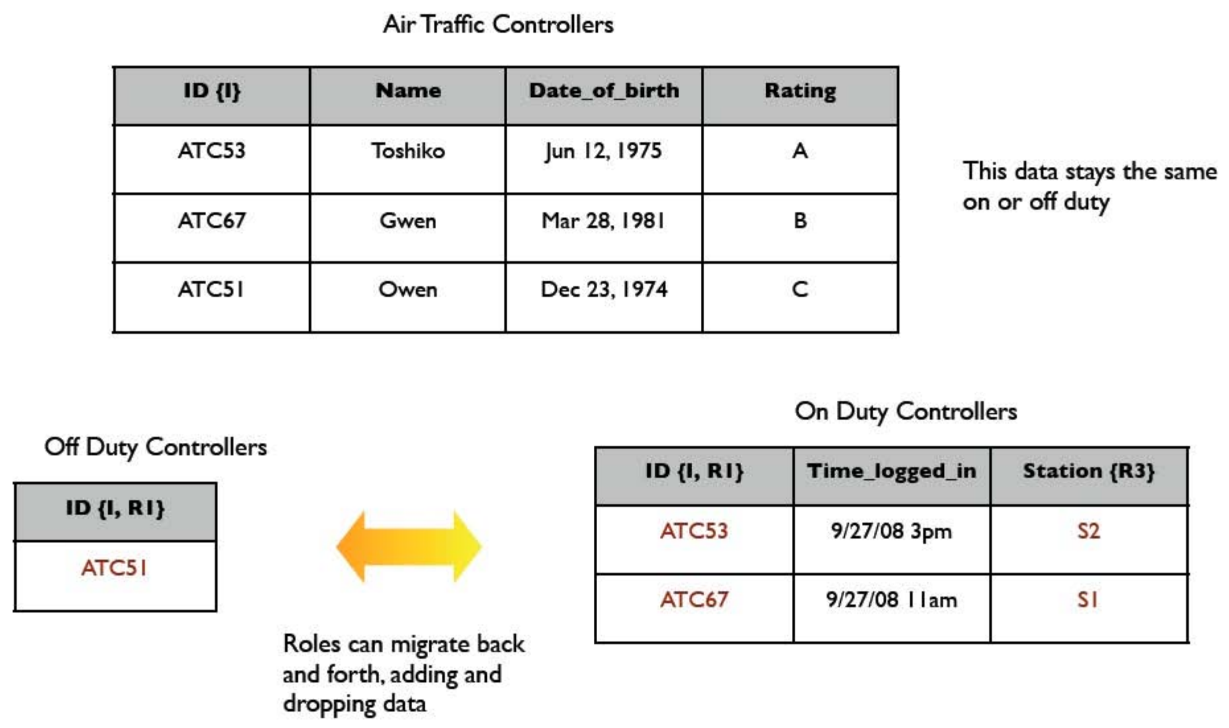
Off Duty Controller

ID {I, R1}
ATC51

Oddly, there is no data kept for Off Duty Controllers. None that we know of yet, anyway. The main value of this class is that it shows that login times and station IDs are NOT maintained for Off Duty Controllers. This is a trick that keeps us from needing "not applicable" values in our tables.

What's wrong with "not applicable"? If you permit these non-values, you are asking for if-then logic in your code to treat the special cases. You are also demanding storage space that the application doesn't really need.

Also note that when an ATC goes off duty, the Station and Time_logged_in data will be discarded (though possibly archived). Since an ATC can't be both on and off duty at the same time, when an ATC object migrates, it acquires or loses data as evident in the populated tables below.



Air Traffic Controllers

ID {I}	Name	Date_of_birth	Rating
ATC53	Toshiko	Jun 12, 1975	A
ATC67	Gwen	Mar 28, 1981	B
ATC51	Owen	Dec 23, 1974	C

This data stays the same on or off duty

Off Duty Controllers

ID {I, R1}
ATC51

On Duty Controllers

ID {I, R1}	Time_logged_in	Station {R3}
ATC53	9/27/08 3pm	S2
ATC67	9/27/08 11am	S1

Roles can migrate back and forth, adding and dropping data

Role migration

A distinction is thus drawn between an ATC object which is the sum of its general and specific data, represented at any one time by two class instances.

Control Zone

Name {I}	Traffic	Controller {R2}
CZ1	12	ATC53
CZ2	4	ATC53
CZ3	8	ATC67

A Control Zone is a region of air space managed by a single ATC. The rule we want to capture is that each Control Zone must, at all times, be handled by an ATC.

For each Control Zone we have a unique name, a quantity of traffic and an assigned ATC. You can probably imagine other useful attributes like location, volume, etc, but I am trying to keep this teaching example simple!

So now we can answer questions like “What is the quantity of traffic managed by ATC53?”

Duty Station

ID {I}	Location	Capacity
S1	Front	20
S2	Front	45
S3	Center	30

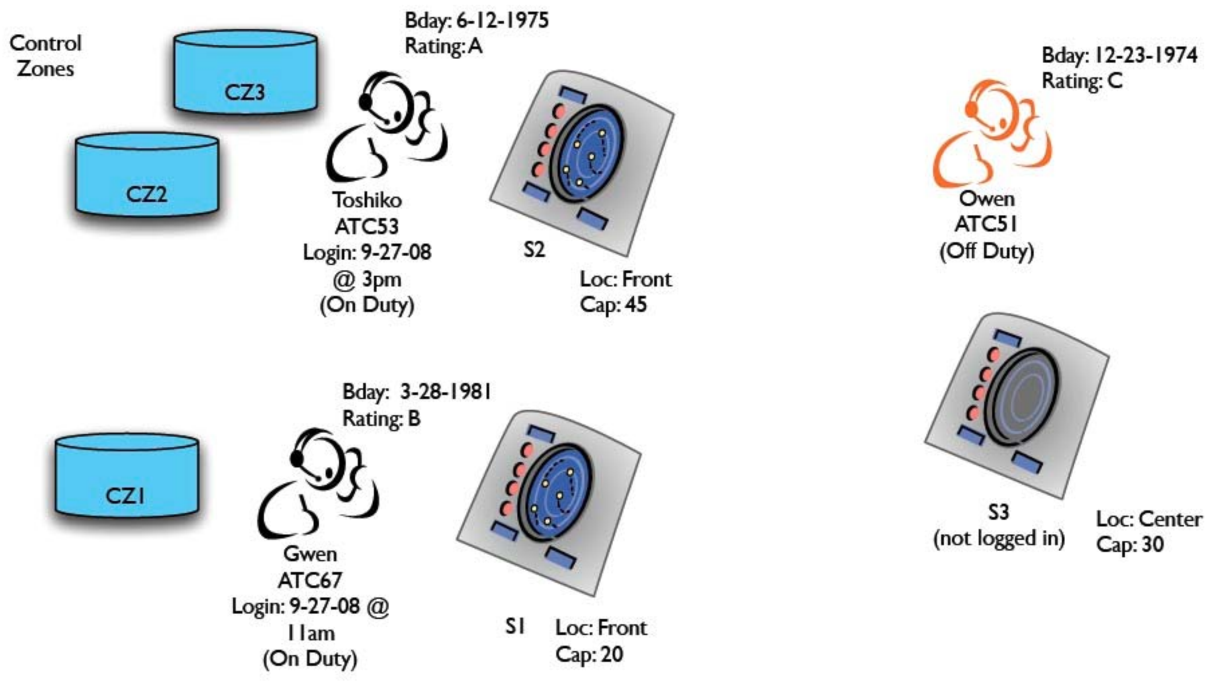
Each Duty Station has a unique identifier, a general location in the facility and a maximum amount of traffic (Capacity) that it is certified to manage. There are no referential attributes here since we already have one in the On Duty Controller class. So if we ask a question like “How old is the person logged into station S2?” we can handle it. Just take the value “S2”, scan through the On Duty Controller table to locate the row. If it exists, go into the ATC table, find the row containing “S2”, and get the value out of the Date_of_birth column. Compare to the current date (should be available as a core system service), do the math and you have the age value.

As mentioned earlier, these access steps are only relevant when reading or executing the model. When you add state and action models, you can execute the models in a simulator that manages state transitions, event queuing, collaboration and data access/computation. Once you get everything working, you can generate or write the code. It is up to the programmer or model compiler to devise an efficient way to glue the pieces together. This sets up a nice division of responsibility with the modeler/analyst saying what data must be connected on *any* platform and the programmer/model compiler finding clever ways to implement the connections for specific platforms.

This is not as complex as it sounds since a single systematic translation pattern may be applied (along with a few configuration parameters for local customization) across numerous model elements. A handful of such patterns will suffice to handle most, if not all of the code. So there is no need to lovingly handcraft the code for each individual class.

An illustrated scenario

The tables are a nice way of formalizing the abstract structure of data. But an illustration can be helpful in testing to see if those tables can accommodate a specific real world scenario.



Illustrated ATC Scenario

The illustrated scenario shows three ATCs, two on duty and one off duty. Each On Duty Controller is logged into a single dedicated Duty Station. We have recorded the login time for each of these ATCs. Each Control Zone is being managed by a single On Duty Controller. The Off Duty Controller is not logged in and has no login time recorded. One Duty Station is sitting inactive. Neither the Off Duty Controller nor the inactive Duty Station has any association with Control Zones. It seems our Good Model can, in fact, accommodate a realistic scenario.

This style of diagram, which purposely avoids any UML notation, is one useful tool for avoiding the dreaded analysis-paralysis. When you limit yourself to a bland palette of boxes, lines and stickmen, it is easy to get absorbed in your abstractions and lose sight of interesting situations that break the rules. Exclusive focus on your model elements, ironically, makes it easy to miss patterns in the real world data that may yield clever abstractions.

So to keep from getting caught up in model hacking (a constant struggle!), I alternately draw a freeform scenario diagram, then build a UML model, then compare them. Validation works in both directions. Can I populate my tables with the data in the scenario illustration? Can I generate a convincing scenario illustration using only the data in my tables? If I discover a new scenario, I check to see if the data will fit the model. If I extend the model, I may draw an updated scenario to see if it makes sense. Going back and forth in this manner will repeatedly reveal flaws in my thinking and ultimately lead to a solid, articulate model. As a nice side effect, I leave a trail of useful documentation! Another great benefit is that users and application experts will give me much more detailed and useful feedback on a scenario illustration than on a model or sequence diagram. Unfortunately, existing UML tools support only the line-box-stickman aspect of modeling and analysis. A huge swath of analysis tooling is yet untapped!

So invent your own notations and icons, steal clipart, draw freely and use the UML notation to crystallize your abstractions.

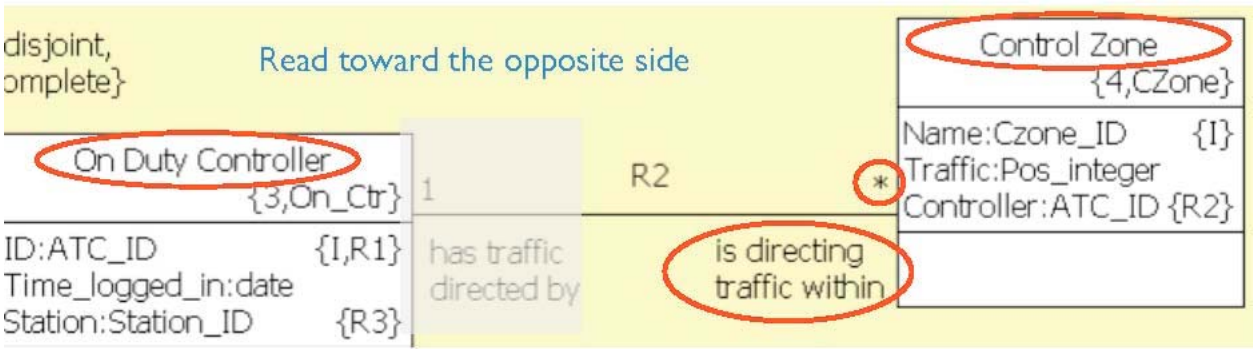
Air Traffic Control Relationships

So far, our focus has been on the classes but there remain two critical associations in the Good Model.

R2 - is directing traffic within

If you look at R2 on the diagram you may notice that the style of naming the roles is a bit different than what you may typically see on common UML class diagrams. This is intentional, and you'll see that I use the common, and less descriptive style, on the Bad Model example.

To read an association named using verb phrases, start on one side of the association and read the phrase, multiplicity and class name on the opposite side as shown.



On Duty Controller is directing traffic within zero or many Control Zones

In plain English, we have "an On Duty Controller is directing traffic within zero or many Control Zones". So we see that it is possible for an ATC to be on duty, but not handling any Control Zones at a given time.

The other way around we have "a Control Zone has traffic directed by exactly one On Duty Controller". So at all times, a given Control Zone is being handled by some On Duty Controller. And only one at a time!

R3 - is logged into

Here we see that an On Duty Controller is logged into exactly one Duty Station. So it is clear that you can't be On Duty unless there is an available Duty Station and you are successfully logged in. From the other perspective, a Duty Station may or may not be operated by an On Duty Controller at any given time. You could also say "is operated by zero or one On Duty Controller" - same thing.

Rules Expressed by the Good Model

Now that we've walked through the model elements, the important thing is to take stock of the rules and constraints expressed, not just by individual elements, but by all the elements taken in combination!

1) An Air Traffic Controller is either an On or Off Duty Controller at any given moment. {R1}
2) An On Duty Controller must be logged into a single Duty Station. {R3}
3) At any given time, a Duty Station may or may not be operated by (logged into by) a single On Duty Controller {R3}
4) A Control Zone must have its traffic directed by exactly one On Duty Controller at all times. {R2}
5) An On Duty Controller may or may not be directing the traffic in one or more Control Zone at all times. {R2}

What Behavioral Constraints are Expressed in the Good Model?

As you can see, the multiplicity and phrase on each side of an association is critical to establishing precise rules. Now let's ask some behavioral questions. To get the full benefit, put the Good Model in front of you and work out the answer to each question on your own. No cheating!

1) What must happen when an Off Duty Controller becomes On Duty?
2) If there is only one On Duty Controller left, can he or she go off duty?
3) If every Control Zone is being directed, can another Off Duty Controller log in?
4) If there are 5 On Duty Controllers and 5 Duty Stations, what must be done to take a Duty Station off line for maintenance?
5) If there are 3 Control Zones and 3 On Duty Controllers what is the maximum number of Control Zones handled by the same On Duty Controller?
6) Assuming there is at least one instance of Control Zone and only one On Duty Controller, can that On Duty Controller go off duty?

The Answers

1) To become On Duty, it is necessary to log into an available Duty Station. The Off Duty Controller instance will be deleted and replaced by a new instance of On Duty Controller referring to the same ATC instance. In embedded systems, creating and deleting in the model is sometimes implemented by preallocating a region of memory and flipping a bit to indicate whether an instance is really there or not. So there are other ways to implement creation/deletion other than constructors and destructors.

2) It depends. Every Control Zone must be directed. Assuming there are one or more instances of Control Zone, the answer is "No". It won't be possible to migrate the On Duty Controller instance without breaking the left side of the R2 association, unless there are zero instances of Control Zone in which case the answer would be "Yes". If there were just one other On Duty Controller, the Control Zones could be handed off first, but there isn't in this case, so it's not an option.

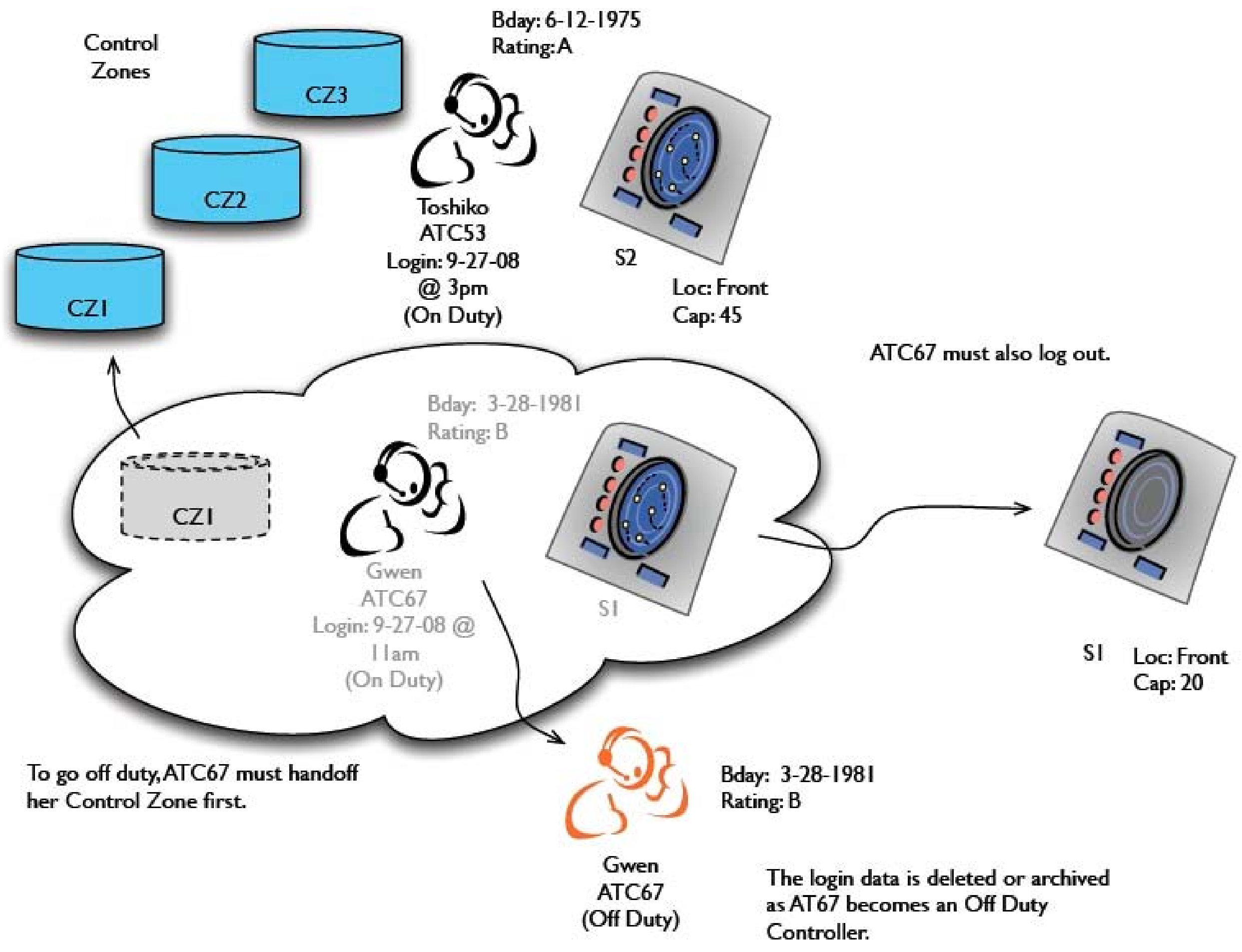
3) Yes. An On Duty Controller is not required to direct any Control Zones. Consequently, the number of On Duty Controllers is only limited by the availability of Duty Stations.

4) Since there is an equal number of Duty Stations and On Duty Controllers we can assume that each Duty Station is in use. So we must log out an On Duty Controller, but to do that, we must make take the ATC off duty. And to do that, we must first ensure that all Control Zones directed by that On Duty Controller are first handed off to some other On Duty Controller.

5) Three. It is okay for an On Duty Controller to have zero Control Zones, so one of them could be directing all three. The other two On Duty Controllers would not have any Control Zones assigned. We just need to ensure that each Control Zone is being directed.

6) No. (Some Off Duty Controller must first become on duty, then all Control Zones must be handed off).

Here is an illustration of some behavior required by the Good Model when an On Duty Controller goes off duty.



ATC67 migrates from on to off duty

Advertisement - Software Development Tools - Click on ad to reach advertiser web site

Software Development Tools

http://www.softdevtools.com/

Your reference for software development tools industry news and for searching open source or commercial tools in every area of application development with more than 1200 tools listed and classified.

The populated ATC tables are updated as shown:

Air Traffic Controllers

ID {I}	Name	Date_of_birth	Rating
ATC53	Toshiko	Jun 12, 1975	A
ATC67	Gwen	Mar 28, 1981	B
ATC51	Owen	Dec 23, 1974	C

Off Duty Controllers

ID {I, R1}
ATC51
ATC67

On Duty Controllers

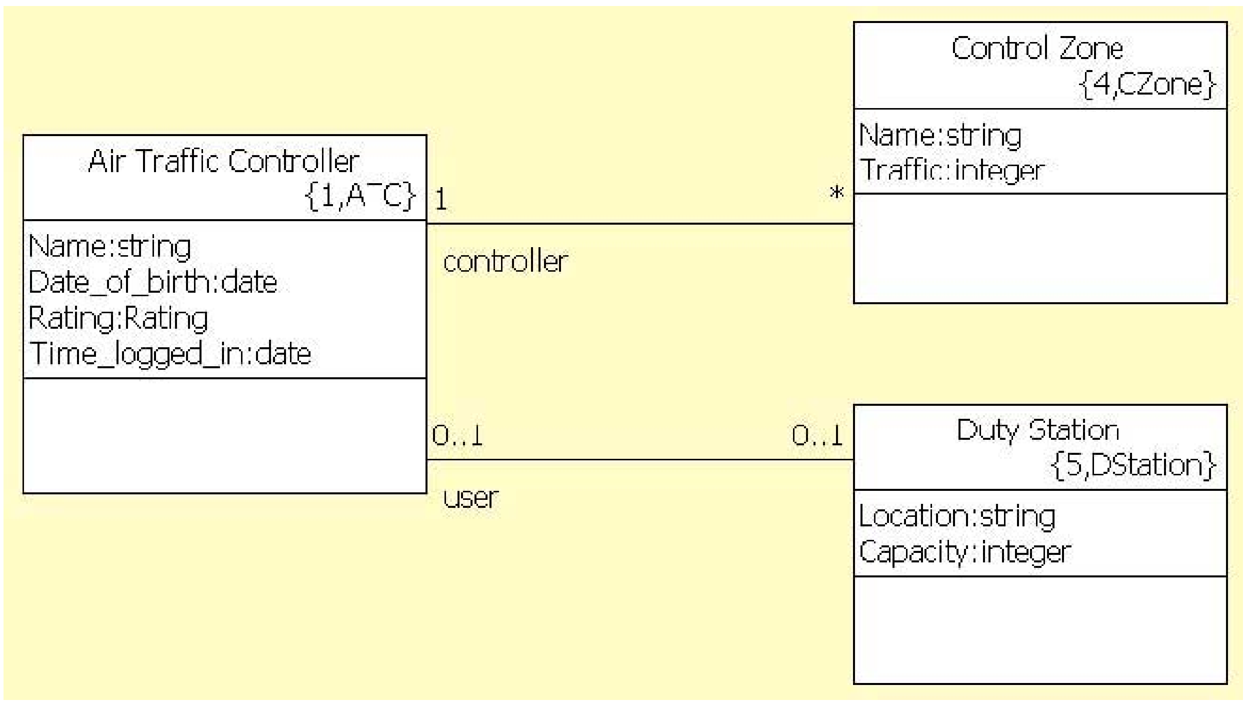
ID {I, R1}	Time_logged_in	Station {R3}
ATC53	9/27/08 3pm	S2

ATC67 migrates from on to off duty

The goal is to create a model that allows legal data sets only. After all, if only legal data can be entered into the data structures, there is no need for code to check for the consequences of bad data! More restrictive class models mean less code, less testing, and higher quality from the get-go. In practice, this can be an elusive goal, but the pursuit causes difficult questions to be asked and answered. As we will see, the Bad Model fails in this regard. On to the Bad Model!

A Bad Class Model

Now let's take a look at the type of UML class model seen on numerous projects and unfortunately encouraged by many books on UML.



It’s the same application, but a very different model. A superficial comparison will reveal the following differences.

1) Fewer classes and relationships
2) Shorter and incomplete names on associations
3) No referential attribute or identifier tags
4) Less precise (implementation oriented) data types on attributes

A careful look, taking all the elements together, reveals that some application rules are missing and even stated incorrectly. First let’s take stock of the superficial differences and then dive into the deeper problems affecting behavioral constraints.

Fewer Model Elements

There are fewer classes in the Bad Model because on and off duty roles are not modeled. One consequence is that we will have a meaningless value for Time_logged_in for each off duty ATC. This will introduce some if-then logic to the actions and/or statecharts. Also this model is telling the programmer or model compiler to reserve space for an unnecessary attribute. Now this may not be a big deal with a handful of ATC’s, but what if we had thousands? Still it might not matter much, but what if we scattered not-applicables across a multitude of classes? In some systems it is not unusual to have hundreds of thousands of instances. So this is just one small way in which a more compact model can lead to less efficient code. In practice, there are many times when a larger, more detailed class model carries the DNA necessary to spawn a tighter implementation.

Naturally, less is better, but only as long as it does not come at the cost of expressing application rules. Modeling is all about analysis which means “taking things apart”. Design and implementation, on the other hand, focuses on synthesis, “packing things together”. A modeler performing object oriented analysis strives to divine and expose all of the application rules using as many components as required. The programmer or model compiler maps and repackages the analysis elements as necessary to yield an efficient design on the target platform. This repackaging must account for every modeled rule, though it is perfectly acceptable to reduce the total number of elements in the process. That’s what design is all about - clever packaging!

It is important to understand that the analyst’s and implementer’s goals are often contradictory. A model that attempts to satisfy both purposes simultaneously usually compromises each. You end up with cramped analysis and bloated implementation which is one reason why good programmers often avoid UML. Our goal in keeping design out of the models is to get the best of both worlds.

Shorter Names

The common role naming style is used to label associations the Bad Model. Roles are okay if you are creating a class diagram as a more or less direct picture of your code implementation. For analysis purposes, however, verb phrases are much more effective.

In the Bad Model, R1 says that an ATC plays the role of “controller” with respect to zero or many Control Zones. Since it is hard to think of an opposite role “controllee?” it is just left out. The “think of a role name game” is often a waste of time that leads to some pretty stupid names. You don’t have to play word puzzles with verb phrases, though you do need to think about what you really mean.

In fact, since the role name often matches the associated class name, you can usually drop them altogether without affecting a model's expressiveness. An Air Traffic Controller is a controller? Thanks, that clears things up!

In fact, it is difficult to tell from the Bad Model whether we mean that an ATC is assigned a Control Zone on an ongoing basis (like a reserved parking space or a favorite coffee cup) or if we are talking about the current moment. Note that the Good Model verb phrase pair "is directing traffic within" / "is having traffic directed by" clears up the temporal confusion in addition to explaining what "control" really means.

By placing a precise verb phrase on both sides of each association, the analyst is forced to consider the multiplicity and conditionality carefully on each side. Since this is where many of the rules are expressed, this is critically important. Generic, all-inclusive verb phrases such as "contains", "is a group of", "has" and my favorite "is associated with" are to be avoided. You need only google "composition vs. aggregation" to get a sense of the frivolity and futility in using generic terms to express precise associations. Which statement tells you more? Memory Block "is partitioned into mutually exclusive" 1..* Region or Memory Block "is a (pick one - aggregation / composition) of" Region?

When you say that an ATC "is directing traffic within" <how many?> Control Zone(s) you are forced to consider the application rules in detail. You should realize at this point that it is 0..* and that the zero case is due to being off duty. Rephrasing as class "On Duty Controller" "is directing traffic within" <how many?> Control Zone(s), you hit on a definite multiplicity of 1..* and nail down an important rule. Flipping the verb phrase from active to passive, you get Control Zone is having its traffic directed by exactly one On Duty Controller. There must always be one and he or she must be on duty according to the application rules. Verb phrases guide a model to increased precision.

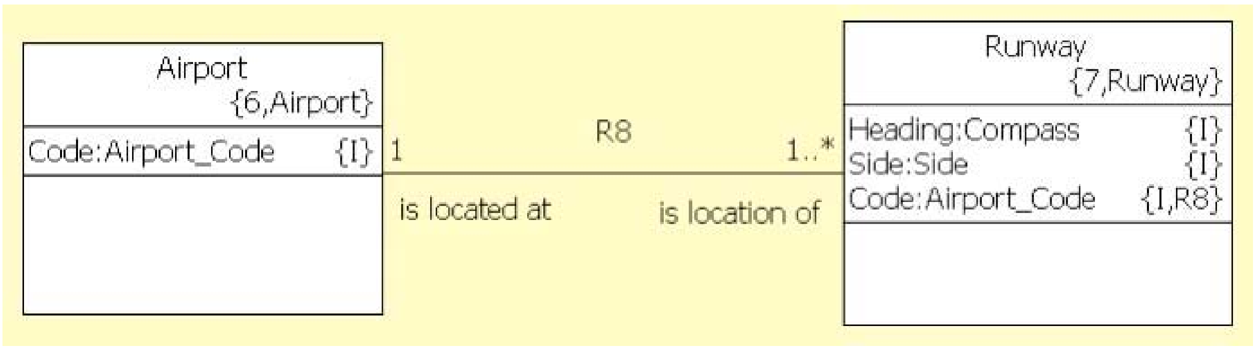
Why all this fuss when the phrases themselves aren't actually translated into code? Actually, they may appear as concatenated names or appear in comments. But the model compiler is not smart enough to comprehend the text in the verb phrase and choose any kind of design based on it. Probably a good thing, too. After all, it's the multiplicity that directs the choice of data structure and access implementation. Ah, but how did you arrive at the correct multiplicity? Models with generic association names almost always belie the true nature of the association and obscure interesting boundary conditions leading to incorrect multiplicity which begets incorrect code.

No Identifier or Referential Attribute Tags

In this particular example, neither tag type plays an instrumental role, so I won't dwell on them. The referential attributes simply serve to show that the data is truly connected in table form. Modern model compilers assume the existence of referential attributes and can manage them behind the scenes. That said, there are a few simple techniques where you can propagate and merge referential attributes across multiple relationships to express powerful constraints. I do this a lot, so by default, I retain the referential attributes.

We assume that each instance of a class is automatically unique. In other words, the code will be generated such that each row of a table corresponds to a uniquely selectable entity. Consequently, we don't need to slap an {I} attribute on every class. Thus if you have a class called "Thing" you could give it an artificial attribute Thing.ID {I}, but it is not necessary. That's true only for artificial identifiers.

On the other hand, real world uniqueness constraints should always be expressed. Consider the case where we model runways at multiple airports. You can't have two runways with the same heading + side at the same airport. (Two 28Rs for example!) How do you express this constraint? Here's a model that does it by incorporating a referential attribute as an identifier component.



Incorporating referential attributes into an identifier to express a real world constraint

This model says, thanks to the {I} and {R} tags in the Runway class, that a unique instance of Runway can be selected by providing a Heading, Side and Airport Code, 28L at SFO, for example (Runway 28 Left at San Francisco International). So you can combine the identity and referential tags to express a fact about the real world. Namely: Multiple runways may have the same Heading + Side, but not at the same Airport.

Less Precise Data Types

The Good Model uses tightly defined application data types whereas the Bad Model relies on loose implementation types. Consider the Control_Zone.Name attribute. In the Good Model it is defined as type "CZone_ID". You can see from the illustration that Control Zones are named with "CZ" followed by an integer value, thus "CZ1, CZ2, etc". Arguably, this can be accommodated by the loose string type. But when we say, in the model, that we require string, what are we asking of the implementation? If the programmer/model compiler works from the more specific application type, he/she/it can choose a tighter implementation type if necessary.

Going back to the single class model, we have a data type called "Altitude". This could be defined as the amount of meters in the range 70000..-400 with a precision of .01, let's say. That would be more precise with respect to the application reality. Naturally, a programmer may choose to implement this as a float in C or whatever type is relevant given the target platform.

Again, it comes down to the same principle. The model should express the application's true requirements, as minimally and precisely as possible without telling the programmer how to write code. You want the model to shrink-wrap the application reality as tightly as possible. It is also okay to expand your application reality to allow for future requirements, as prudent given your project reality! If the current application handles fixed wing aircraft only, but helicopters requiring helipads instead of runways will be added in the future, it might make sense to rethink the definition of runway and landing surface. Whether you fatten up the application rules or not, you still need to build a precise model of whatever reality you define. Leaving the model vague is never the right way to accommodate future requirements - it's just plain laziness. (Plane laziness? sorry...)

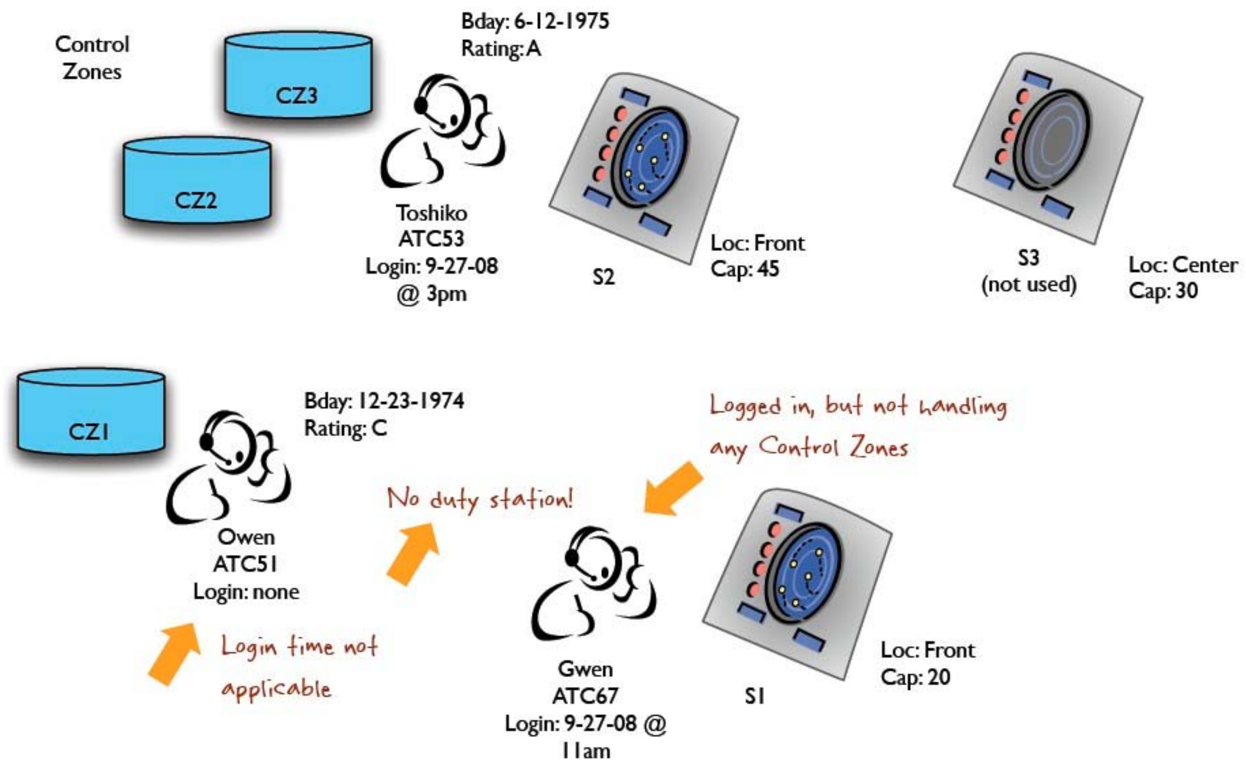
Even when I allow an attribute to be a loosely defined string such as a name, I might use the data types "Long Name" and "Short Name". The first might be defined as a string up to 80 characters while the other might be up to 10 characters. They could be targeted to the same or different implementation string types.

Behavioral Constraints not Expressed and Questions Unanswered in the Bad Model

Now let's take a deeper look and consider the model as a whole. What does the Bad Model say about the ATC application behavior and is it correct? In the Bad Model, the "controller" association is 1:*. The * side permits zero. This glosses over the subtle distinction between an on duty controller with no current Control Zone assignment and an ATC who is off duty. Moving on to R2 in the Bad Model, we see that it is 0..1:0..1. This noncommittal association covers both off duty controllers who aren't logged in as well as Duty Stations not in use at the moment. So we've lost the constraint that while on duty, an ATC must be logged in. If an ATC is not using a Duty Station, can he or she still be controlling a Control Zone? The model says "yes" since the two associations on the Bad Model are completely independent of one another. If an ATC is not controlling any Control Zones can he or she still be logged into a Duty Station? Both Good and Bad Models permit it. And, according to the application rules, this is okay.

With the single class "Aircraft" table example, we considered the questions that could be answered by a model. Try asking the Bad Model the question "Which Air Traffic Controllers are off duty right now?" The Bad Model cannot really answer this question. It can tell you which ATCs are not handling any Control Zones. But those ATCs may be on duty. Can you fix it by adding a status attribute to the ATC class in the Bad Model? Sure, but you still have no protection against an ATC with the value "off duty" handling a Control Zone. The status attribute solution also necessitates action language to manage the constraint. That's more code and more testing, all due to one status attribute. A single status attribute now and then is not so bad. But when abused, and it so often is, the resultant complexity in the state and action models can be dramatic

Here is one example of an illegal scenario allowed in the Bad Model:



Bad Model allows an ATC to work without a Duty Station

The picture above looks okay with the exception of ATC51 (Owen) who is handling Control Zone CZ1, but is not logged into a Duty Station.

Advantages and Disadvantages of the Bad Model

[+] In defense of the Bad Model, it does not *forbid* legal data populations. Data representing an Air Traffic Controller logging in and out of a Duty Station is accommodated. But the same is true for data illegally showing that an on duty Air Traffic Controller is directing traffic in a Control Zone.

[-] The problem is that some illegal configurations are *also* possible. Unless action language and/or code is written to carefully enforce these constraints bugs, will emerge down the road.

[+] Another advantage of the Bad Model is it probably took less time to build.

[-] Sure, less thinking went into it. The time saved will be eaten up writing action language to define and handle the constraints or fixing it in the code. And if any of the constraints slip through those cracks, the time will be spent in testing and debugging.

The purpose of modeling and analysis is to expose and evaluate requirements. We analysts break concepts apart while programmers (or model compilers) compact them down into an efficient ball of code. The class model is a powerful tool when it comes to exposing rules. Even without knowing much about class models, it is not difficult to walk an application expert through the Good Model and get valuable feedback. Walk them through the non-committal Bad Model and you'll just get tepid nods of approval.

The Good Model, on the other hand, might raise questions as to whether or not a Duty Station might be shared and to the specific circumstances where this would occur.

The right tool for the job

To be an effective analyst you want to reach into the UML toolbox and retrieve the best tool for the job at hand. Class models are the most powerful tools for expressing rules and constraints directly in data structure. State models are great for formalizing sequence and synchronization. A state model, for example, is ideal for sequencing the actions required to migrate between on and off duty status. E-mail a request to me at leon_starr@modelint.com and I can send you the state model and action language for the Good Model. (This is my way of finding out if anyone has actually read this far!) Action language is useful for describing data flow, computation and data access.

By deferring rules and constraints to less effective tools (states and actions), you end up creating more work for yourself. Worse still, neglected rules and constraints tend to get buried in complex actions where they are not as easily seen by application experts, if not entirely forgotten. One of the main reasons to model is to resolve potential flaws early by tweaking a relationship or attribute rather than later by overhauling an entrenched implementation.

Why not just express constraints with OCL?

There is an object constraint language (OCL) included in the UML. Having a language dedicated to constraints is a really good thing, but there are a few downsides.

1) OCL is not supported by many tools

2) OCL can be difficult to read and write
3) A constraint written on the side can be less reliable than one integrated into the data structure
4) Constraints written on the side can result in more code to write and test

Here is an example [1] of the OCL corresponding to the identifier {I} tag:

```
context <class> inv:
        <class>.allinstances()->forAll( p1,p2 |
        p1<> p2 implies
           p1.<identifier> <> p2.<identifier> )
```

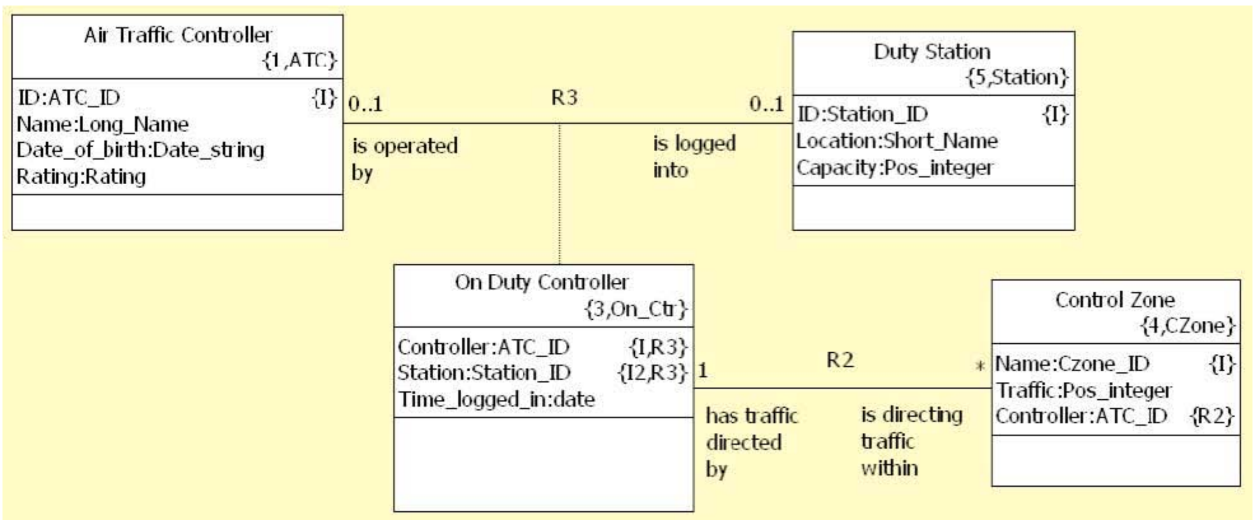
It's nice to have both a tag for the class model as well as a detailed expression of the constraint.

OCL can be useful for expressing constraints not easily shown on a class model, like ensuring that the of value assigned to one attribute is less than a maximum value specified by another attribute, for example. So using OCL to fine tune the constraints on an articulate class model can result in a powerful combination.

But there is no better way to enforce constraints and prevent bugs than to define data structures that will not accept bad data in the first place. Building a Bad Model and expecting to express all your constraints in OCL is going to be rather difficult. This is mainly because a Bad Model won't offer the detailed vocabulary (classes, attributes and relationships) to which the OCL would refer!

Making the Good Model more concise

As promised we can condense the Good Model a bit without sacrificing any application rules.



A more concise Good Model

All I've done here is replace the generalization relationship with an association class. Since there are no attributes or interesting behavior on the Off Duty Controller, we can eliminate the class. Now an instance of On Duty Controller is created as a consequence of linking an ATC object to a Duty Station object. Upon unlinking, an ATC from a Duty Station (logging off), the representative On Duty Controller instance disappears along with the Time_logged_in attribute value and any Control Zone links. Note that the multiplicity on R3 is 0..1 on both sides. So, at any given time, an ATC may or may not be logged in. From the perspective of a Duty Station, it may or may not be available.

So why didn’t I just do it this way in the first place? There are a couple of potential benefits to the original Good Model. The Off Duty Controller subclass serves as a nice placeholder in case we do discover attributes or behavior unique to that role. Also, if we discover an association that applies only to an Off Duty Controller, we would have an anchor for it. Also, the original example is a bit more intuitive since the subclasses mirror the primary states of an ATC. The limited scope of this teaching example favors the association class solution, but real world complexity may necessitate the generalization solution.

Some suggested techniques

Any time you see a relationship between two classes with * (zero or many) or 0..1, you have conditional associations. There’s not necessarily anything wrong with that, but it should make you ask whether or not the conditionality is due to changing roles or circumstances. If so, consider specializing or using an association class. Look, especially, for associations and attributes that would be relevant only to certain roles or circumstances.

If you have attributes that can take on “not applicable” values, then you should probably refactor your analysis a bit. Perhaps a generalization or association class may help create a better home (class) for the conditional attribute. Think of a class as “The definition of a set where each member of the set exhibits the same behavior and has the same attributes and associations.” So if the set you are defining has any “special” members, refactor! Strict adherence to this principle will often lead to more classes and relationships in your class model. But it stands to reason that a class model with an expanded vocabulary can be more intelligent. And you, yourself, will use this richer vocabulary to phrase more penetrating questions about the application requirements and make the model, and ultimately software, even more intelligent!

For every association use verb phrases instead of role names. Name both sides of every association as precisely as possible. This will force you to consider the multiplicity on each side more carefully and, again, lead you to a clearer understanding about the deeper truths in your application.

Tag all real world identifiers (tail numbers, airport codes, license plates, serial numbers, etc). Look, especially for multi-attribute identifiers (airport + heading + side), (file cabinet name + drawer number) and so forth. Incorporate referential attributes in these identifiers. You will be surprised how many important constraints can be discovered. Sketch non-UML scenarios with specific instances and data values corresponding to the classes, attributes and relationships in your UML model. Use these to solicit feedback from the application experts and to explore boundary conditions.

Summary

The value of creating UML class models depends greatly on what kinds of models are being built. Imprecise class models that do not express key constraints are often the result of critical misconceptions about the purpose of class modeling. One of these is that class models are static and therefore do not play a role in defining overall system behavior. Another is that analysis, design and implementation goals can somehow be reconciled within the same model. You can, of course, just use UML to draw pictures of your code, but, if you’re going to do that, why not save time and just write the code?

The good/bad model comparison was intended to illustrate some ways in which a good class model can express detailed application rules and shape and constrain overall system behavior. If-then logic and other behavioral switches can often be embodied directly in data structure. This reduces the complexity of the statecharts and actions and leads to less code and testing.

Rather than think of a UML class in implementation terms, like a Java or C++ class, a UML class is defined, for analysis purposes, as a set of things with the same characteristics, behavior and relationships formalized in a relational table structure. This table structure may be implemented in a wide range of platform data structures not necessarily resembling tables.

This frees the analysis from platform specific considerations. The analyst is free to focus on the application rules that must be enforced in any implementation. The result is an application rule base that remains stable as the underlying platform evolves. Another benefit is that the application models can be retargeted to migrating or spin-off products that deploy the same rules on different technologies.

There are a number of analysis techniques that lead to useful, rule expressive class models. One is to balance abstract modeling with concrete scenarios using detailed, non-UML illustrations. Another is to use verb phrases to name all association sides articulately as a way of obtaining precise multiplicities.

This approach should yield practical and compelling benefits from UML class modeling.

Resources

The ATC example, while only shown here as a class model, is available with states and actions in a fully executable form if you have either the Mentor Graphics BridgePoint or Kennedy Carter iUML tools. If you would like a copy, please send me an email. If you don’t have either of these tools available, I can send you PDFs and/or text files with the complete set. Feel free to use them for commercial or noncommercial purposes as long as you retain the copyright information and credit the author.

[1] Executable UML, A Foundation for Model Driven Architecture, Mellor-Balcer, Addison-Wesley, 2002, ISBN 0-201-74804-5

[2] Executable UML, How to Build Class Models, Leon Starr, Prentice-Hall, ISBN 0-13-067479-6

[3] Model Driven Architecture with Executable UML, Raistrick, et al, Cambridge University Press, ISBN 0-521-53771-1

[4] Databases, Types and the Relational Model, C.J. Date, Hugh Darwen, Addison-Wesley, ISBN 0-321-39942-0

When Good Architecture Goes Bad

Mark Dalgarno, mark @ software-acumen.com
Software Acumen, www.software-acumen.com

Every developer eventually encounters it at some stage in his or her career – the code that no one understands and that no one wants to touch in case it breaks. Sound familiar?

But how did the software get that bad? Presumably no one set out to make it like that? The answer is that the software is suffering from ***Software Erosion*** **–** the constant decay of the internal structure of a software system that occurs in all phases of software development and maintenance

At the architectural level, Software Erosion is seen in the divergence of the software architecture **as-implemented** from the software architecture **as-intended**. Note that when talking about the architecture as-intended I'm not speaking here about the initial planned architecture of the software system. Software architectures should evolve over time – this is to be expected as new requirements emerge – so the intended architecture is what your **current** conception of the architecture is. With software erosion what we're talking about are **unintended** modifications or **temporary** violations of the software architecture.

The problem with software erosion is that its effects accumulate over time to result in a significant decrease in the ability of a system to meet its stakeholder requirements. Unless you take steps to actively pinpoint and stop software erosion it will gradually creep up on you and make changing the software further significantly harder and less predictable. In the worst case it could lead to the cancellation of the project or, for particularly significant projects, the closure of the business.

Types of Software Erosion

To begin to tackle software erosion you need an understanding of how it typically shows itself. Common types of software erosion include:

- **Architectural Rule violations** e.g. where strict layering between subsystems is bypassed.
- **Cyclic dependencies** – for example **A** calls **B** calls **C** calls **D** calls **A**. This type of dependency can be valid but when it's unintended can lead to very complex, opaque code that is hard to understand and hard to test in isolation.
- **Dead code** – code that once supported part of the software, is now no longer used, but is still cluttering the code base.
- **Code *clones*** – identical or near-identical code fragments scattered across the system. A bug fix or change in one clone instance is likely to have to be propagated to the other clone instances.
- **Metric outliers** e.g. very deep class hierarchies, huge packages, very complex code etc.

A well-known example of software erosion was highlighted in a reverse-engineering experiment on two separate versions of ANT some years ago. ANT V1.4.1 (11 October 2001) and ANT V1.6.1 (12 February 2004) were reverse-engineered and the results were compared.

At the time ANT was built in three layers, from the top-down these were *taskdefs, ant, utils*. In the earlier version these layers were well separated and the *ant* layer was monolithic but small.

In the later version the *ant* layer was still monolithic but had now become very large – making it harder to understand and work with. More problematically a new upward dependency from the lower-level *ant* layer to the top-level *taskdefs* layer had been introduced.

These types of erosion problems lead to code that is hard to understand, hard to modify and hard to test. But how do you know whether you're suffering from software erosion?

Are you suffering from Software Erosion?

Perhaps the first thing to observe is that most projects will suffer from software erosion at some stage unless there is a conscious effort to pinpoint and stop such erosion. Even projects that are relatively short-lived can suffer from it. One example I have heard about involved a software project that had to be scrapped after only 6 months because it had already eroded badly.

There are some common things you can look out for when deciding how badly your software is suffering from software erosion:The time, effort and risk in implementing new functionality increase – productivity and quality decrease and complexity increases. These are very common side effects when software erosion is present.

- No one has responsibility for the architecture and knowledge of the architecture is held by a decreasing number of people.
- No one on the team can tell you (or agree on) what the **intended** or **implemented** architectures are. If you don't have an understanding of either of these then it's very likely that software erosion has occurred and will continue to occur.
- The team hasn't had a stable core membership throughout the software's life. If someone leaves the team then that person's knowledge of the architecture and software leaves with him or her. New people take time to get up to speed on the project, so mistakes are made and the software erodes further. If new people are unlucky enough to be introduced into a team where no one knows what the architecture is or should be, then the software will erode even faster as they make changes to it.
- Little or no refactoring is sanctioned. Refactoring is the way to rollback software erosion once it has been pinpointed. Refactorings that remove architecture violations, eliminate code cycles, prune dead code, consolidate code clones and do away with metric outliers are particularly beneficial, because, by fighting software erosion, they clear the way for other refactorings, for bug fixes and for new features in the software.
- There's pressure to rewrite the software. When software has eroded badly it becomes really hard for developers to work with that software. Every change and bug fix takes significantly longer in practice than it should in theory. The code becomes brittle and so even the simplest change can have unexpected knock-on effects which lead to costly rework. I'll say more on rewriting later.

At a detailed level, software erosion results in problems such as code living in the *wrong* place, layering violations (as seen above in the ANT example), complex cycles insufficient decomposition, big packages etc.

Costs of Software Erosion

It can be hard to measure the cost of software erosion and convey this cost to non-technical people who often have to sanction work to stop software erosion. Even though software erosion causes reduced productivity, reduced quality and increased time-to-market, no one specific point of erosion causes these effects in isolation, rather it is the effect of multiple points of erosion that combine and reinforce each other to cause them.

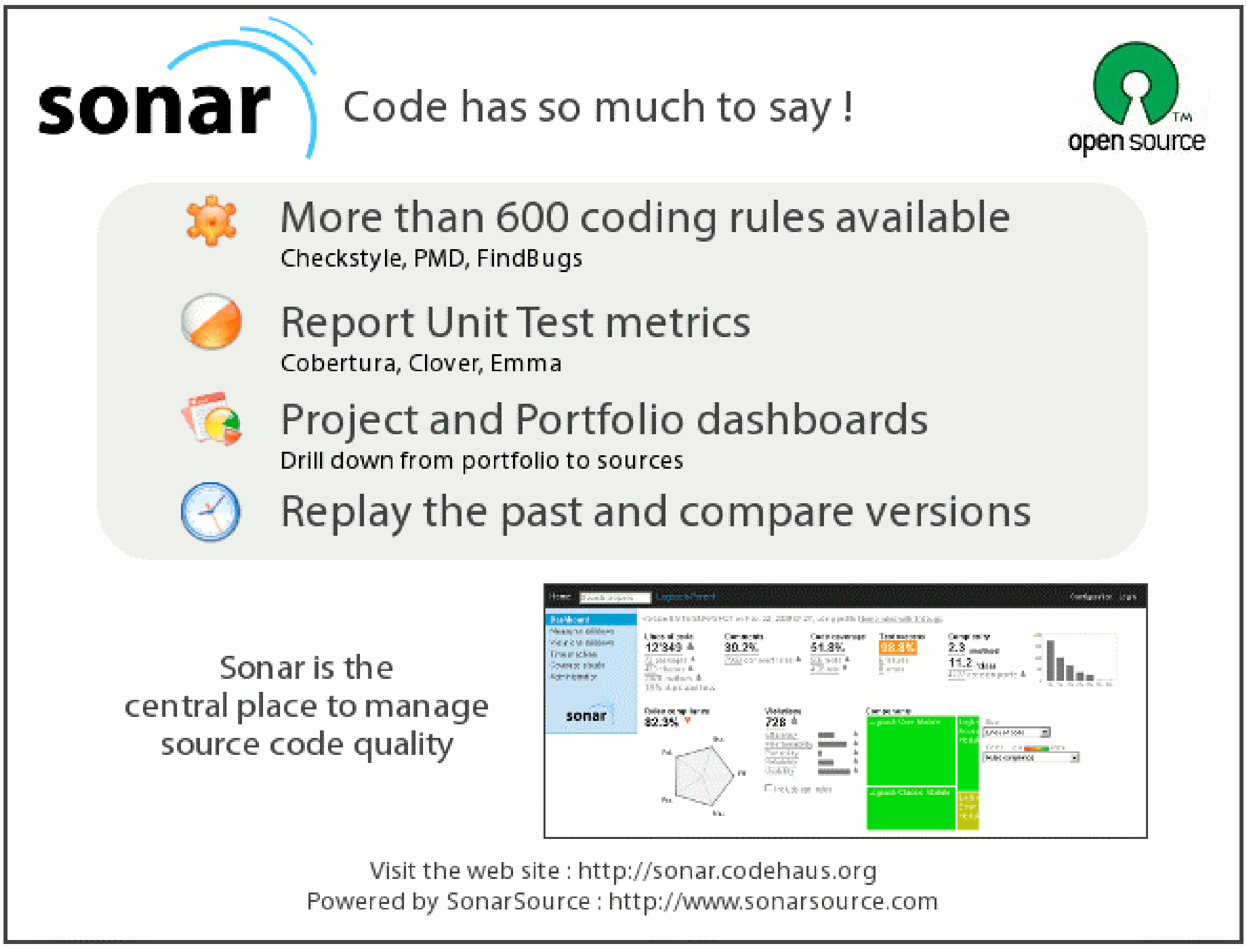
However, a study by the US Air Force Software Technology Support Centre (STSC) attempted to put some rough measure on the costs of software erosion. The researchers took two versions of a mature software system (50k LOC) and asked two different teams to perform the same maintenance task (adding approx. 3k of code) on their respective version. Version 1 was an existing system suffering software erosion. Version 2 was the same system but with the architecture restructured to remove erosion.

The results were staggeringly different. Team 1, working on Version 1, needed over **twice** as long as team 2 to complete this relatively short task. Furthermore, Team 1's results contained more than **eight times** the number of errors than the work submitted by team 2, working on version2. Erosion in a small system such as this still had the potential to lead to significant problems when the software was maintained.

Causes of Erosion

By now you should have some clues as to how software erosion comes about. It does not arise purely spontaneously. Software Erosion comes about through **change**.

Advertisement - Sonar Manages Source Code Quality - Click on ad to reach advertiser web site


sonar
Code has so much to say !
open source
More than 600 coding rules available
Checkstyle, PMD, FindBugs
Report Unit Test metrics
Cobertura, Clover, Emma
Project and Portfolio dashboards
Drill down from portfolio to sources
Replay the past and compare versions
Sonar is the central place to manage source code quality
Visit the web site : http://sonar.codehaus.org
Powered by SonarSource : http://www.sonarsource.com

Pressure for change comes from a variety of sources. The need to add new features to a product to help persuade people to buy it, changes to the environment within which the software is deployed e.g. to support different networking or GUI standards and technical changes, such as the desire to adopt new coding standards all have an impact on the software. Where the initial vision for the software doesn't allow for change, such erosion effects will be seen very quickly.

Software Erosion is also known as software decay or code rot and by similar terms. However, these don't adequately capture the notion that it is forces external to the software that are ultimately the cause of problems within the software. Erosion is not something that just happens to the code without someone actively making such changes. This is why I feel that notion of software erosion more adequately describes this gradual wearing down of the ability to work effectively with the software.

The needs of the business can also contribute to software erosion. Even though deliberately eroding your software causes bigger problems down the line it may be in the best interest of the business to do this for some short-term gain. The problems build up quickly however if the business does this repeatedly without spending time to refactor the eroded code. Every developer is familiar with the 'quick-fix' that becomes a permanent feature.

Real-World Examples of Software Erosion

How bad is this problem in practice? In 2007-08 I decided to investigate this question by running a number of workshops at different software events in the UK and by engaging in some discussions with some software practitioners further afield. At every workshop I ran participants spoke about many different examples of systems suffering software erosion:

- Software with a large number of cyclic dependencies that ended up as brittle spaghetti code.
- Systems where business logic (with associated SQL) was captured in the software's presentation layer – making it hard to replace this layer.
- A software system where the threading architecture eroded so badly over time that the system became unmaintainable and had to be scrapped.
- A single class used as a dumping ground for everything that didn't have a better home.
- A 'cancerous wart' of a software system with ever increasing coupling between modules, packages etc.
- Lots of code clones (copies and near-identical copies).
- Uncontrolled code use – programmers grabbing code, classes and even variables from other parts of the software without any control on what could and couldn't be used – once again led to significant erosion.
- Several examples of *drive-by programming* – team-membership constantly changing, programmers not understanding the architecture and so making mistakes when they coded and then moving onto their next project. One example of *drive-by-architecting* with similar consequences.
- Problems with obsolete software and hardware technology; a lack of skills in these obsolete technologies leading to further decay.
- Sales-driven evolution – where there was no clear roadmap or scope for the software system and so the implemented software architecture inevitably diverged rapidly from the intended architecture.
- Merged companies with different cultures and different principles having to collaborate on a software system leading to decay.

In every workshop all but a few people either were working on projects that had eroded quite badly or had worked on such projects in the past.

Case Study - Outsourcing of a 1MLOC C/C++ system

I outline below a real-world case study in order to get you thinking further about software erosion. My recommendation is to spend 10-15 minutes (either on your own or with a colleague who is also reading this article) thinking about the questions before proceeding to the discussion.

Case Study Project History

A company developed a software system over a number of years. Six years ago the software was transferred to a company-owned outsourcing centre in India where it has been developed since that time. At the time of the transfer the organisation believed that the architecture of the system as intended was well documented and matched what was implemented.

The software is critical and cannot be thrown away easily.

Over time more staff were added to the project to maintain a steady flow of new features. The company has a similar product that is maintained and evolved by 5 developers whereas the Indian department now has 50 developers.

The company recently compared the amount of work done by these two teams and assessed that they delivered roughly the same amount of work.

Present Situation

Acting on this difference in productivity the company compared the architecture from 6 years ago (as the outsourcing took place) against the architecture of the current code and found that many parts of the system have dependencies that are not intended.

The intended architecture was documented, so in theory all involved personnel could have compared actual to as-intended architecture. The initial architecture was probably appropriate for the current system (so it's a good architecture that has gone bad).

The company now intends to bring part of the software back under control in Germany while leaving part under control in India.

Questions

Think about whether it is credible that software erosion led to this significant decrease in productivity? What do you think of the company's proposed solution?

Discussion

The software has been developed over a number of years; the team and their development processes; tools and technologies may have changed during this time. Given we can probably reason that the software has probably been modified a lot before it was handed over and so conclude that it’s likely that the architecture at the time of handover may have eroded.

There was a major personnel change 6 years ago when the project was handed over. The two different organisations will have different cultures, knowledge & skills. It is not clear that these differences will be lessened just because both organisations are part of the same multi-national. This could lead to further erosion.

We also have to consider the reasons for the switch and the way the switch took place. Did the organisation cut costs on the project when the software was handed over? Was there a backlog of work on the project that it was felt the new team could tackle sooner or better? How was the handover done? Did they redeploy the existing team elsewhere or did they fire them? Were people from the old team made available to help people from the new team get up to speed? How much time was the new team given to learn about the software before having to start modifying it? If there was no effective handover and insufficient time allowed for the new team to learn the architecture and the code base then erosion is more likely to have occurred.

We're told that the 'Software is critical and cannot be thrown away'. We're also told that there's been a steady flow of new features Both of these indicate that changes have and will take place implying that erosion could be present. This is confirmed by the assessment that there are a lot of unintended dependencies in the architecture **as-implemented**.

My belief is that **it is credible** that architectural decay contributed to the team's problems but that it cannot be untangled from other issues.

- There have been lots of changes over the years.
- At the time of handover it wasn't clear how closely the architecture as is matched the architecture as intended.
- There were lots of staff changes – how well was the handover managed? – this was initially a comprehension task that needed management and technical support.

Stopping Software Erosion

Stopping software erosion requires management commitment. If managers are only interested in the short-term viability of their software projects then it is hard for developers to get the time and make the effort to tackle the problem. This does not excuse developers from doing what they can to fight erosion but will inevitably make their struggle less effective.

If management commitment is present then the following outline *pattern* can be used to stop software erosion. How you implement the pattern depends on what tools you have available, what domain your project lies in, how mature the erosion problem is etc.

Stopping Software Erosion – a Pattern

1. Start out with a sustainable architecture. – All successful software systems evolve; make sure you have built in flexibility for future **known** changes. Assess your architecture using the most likely change scenarios – where is it flexible, where will it need to evolve? There are always tradeoffs here in the amount of time you can spend in architecture assessment and also in the 'finished' architecture.
2. When implementation starts regularly visualize the architecture as the software changes. Get a feel for how close your implemented architecture is to your architectural vision – maybe you need to change the latter.
3. **Compare** the architecture **as-implemented** to your architecture **as-intended** to see how they differ. With automated support this can be done as part of the software build. This

step does rely on you at least having some *vision* of what your intended architecture is. If you don't have this then you can gradually reverse engineer it from your architecture **as-implemented**. There are now many tools from very basic free ones through to very advanced commercial tools that can help with architecture visualization and checking.

4. Use cycle detection, clone detection, metrics analysis and dead code detection to pinpoint software erosion. Again there are several free and commercial tools that tackle some or these tasks.
5. Refactor the software to remove eroded code.

Stopping Software Erosion – Cultural Factors

As noted above, if top management doesn't support the fight against software erosion then developers have their work cut out to stop erosion. With management support you can create a culture where stopping erosion is valued. This culture is likely to have characteristics such as – an emphasis on regular refactoring, clear assignment of responsibilities, sharing of architectural knowledge and work, frequent communication between the whole group.

In *Designing Maintainability in Software Engineering: a Quantified Approach* Tom Gilb describes one team's 'Green Week' – one week set aside each month to focus on improving their software's maintainability. This proved more successful for the team than their earlier one day a week approach and had the added benefit of making the development team feel empowered.

A few words on rewriting

Before I wrap up I'd like to say a few words about software rewrites. As I noted earlier, pressure from development teams to rewrite software commonly manifests itself when that software has eroded. In the worst case the development team uses the excuse of a possible future rewrite to delay refactoring work to the software. When this occurs, the software continues to erode until it reaches a state where working with it becomes very difficult. Even if a rewrite may once have been avoidable if action had been taken the result is that a rewrite becomes inevitable due to the negligence of the team.

As a developer, when faced with a decision about rewriting some software you should always ask yourself whether you are planning to rewrite it for the right reasons. Is it because you cannot make the software maintainable or is it to get rid of code you haven't tried hard enough to refactor or code that someone other than you has written? Worst still, is it just to get some hot new technology onto your CV?

As a manager ask yourself whether you can afford a rewrite? Do you have the right people with the right skills available for the right length of time? Do you understand the risks of new tools and technologies? Do you understand what you have to build? Are you rewriting the software or building something brand new? Worst still, how long will it be before your competitors catch up? In the Doomsday scenario, can your organisation handle the total failure of the rewriting project?

If you're about to risk an expensive and lengthy rewrite of your software, are you really sure that you've exhausted every approach to fighting software erosion in your current code base?

Summary

Any successful software system is likely to evolve. Unless preventative work is undertaken the software will erode. As the software erodes the cost and risk of further development rises. It’s **rarely too early** to start fighting software erosion. The costs of software erosion start to bite very quickly once it sets in.

There are lots of different things that can be done to stop software erosion – you (just) need to work out what the best value approach is for your particular project. If you are a **manager** then create a culture where fighting software erosion is encouraged and supported. If you don't do this then no one will care about erosion. If you are an **architect** or **developer** then educate yourself about the different causes of erosion and the different approaches for fighting it. If you’re interested in finding out more, or sharing your ideas on stopping software erosion, then please get in touch.

References

See http://www.stsc.hill.af.mil/crosstalk/2005/11/0511SangalWaldman.html for more information on the Ant case study and http://codefeed.com/blog/?p=98 for a brief early Ant project history.

General Background Reading:

Lehman's laws of software evolution: M M Lehman, J F Ramil, P D Wernick, D E Perry, W M Turski, "Metrics and Laws of Software Evolution - The Nineties View," metrics, p. 20, Fourth International Software Metrics Symposium (METRICS'97), 1997

Refactoring in Large Software Projects: Performing Complex Restructurings Successfully, Martin Lippert, Stephen Roock, Wiley 2006

Finding a Partner to Trust: The Agile RFP

Peter Stevens, www.sierra-charlie.com/contact.php
Sierra-Charlie Consulting, www.scrum-breakfast.com

Customers of software development services who want to outsource a software development project face a problem: Traditional methods of selecting a software developer are expensive, time consuming and optimize the wrong criteria. They set the stage for delays, cost overruns, and building software with poor or no return on investment.

Agile methods, including Scrum and XP, have proven successful at creating great solutions that meet the expectations of their sponsors and users. Many organizations would like to apply agile approaches, but the traditional tools for selecting a vendor don't mix well with agile development.

By conceiving the project from the beginning as an agile project, you can outsource projects effectively and agilely. This paper:

1. Describes how one team used Scrum to create an agile RFP
2. Discusses what information should be present in an agile RFP
3. Proposes how to find a partner to trust through a lean, Agile selection process

Advertisement - Sonar Manages Source Code Quality - Click on ad to reach advertiser web site

Background

My customer wanted to develop an application to automate critical work and information flows. These are very complicated, involving direct customers, end customers, the company itself and its suppliers. The company had some expensive experiences with waterfall development processes. Scrum had usually been the solution, so for this project, it seemed logical to conceive the project to mix well with Scrum. But this presented new problems: How do you write an agile, Scrum-Compatible RFP? How do you select a company to implement an agile project?

A Google search [1] was remarkably unhelpful with this problem. The top links all pointed to a paper and presentation from 2003 about combining Use Cases and User Stories in the RFP process. A request to the ScrumDevelopment Group produced no responses [2]. So we were on our own.

A Classical RFP is poorly suited to Agile Development

A request for proposal would typically include items like the template in Box 1. The customer seeks to specify his requirements precisely; the supplier should say exactly what they are going to do, how long it will take, and what it will cost. Because requirements, time tables and deliverables are rigidly specified, this form of contracting does not fit well with agile development processes.

Template for a traditional RFP

1. Introduction and Background
2. Purpose of the Request for Proposal
3. Schedule of Events
4. Guidelines for Proposal Preparation
5. Proposal Submission
6. Detailed Response Requirements
    a) Executive Summary
    b) Scope, Approach, and Methodology
    c) Deliverables
    d) Project Management Approach
    e) Detailed and Itemized Pricing
    f) Appendix: References
    g) Appendix: Project team staffing
    h) Appendix: Company Overview
7. Evaluation Factors for Award
8. Criteria
9. Scope of Work
    a) Requirements
    b) Deliverables

Source: Foundstone RFP Template, http://www.foundstone.com/us/services/foundstone_rfp_template.doc

Using Scrum to create the RFP

My customer assembled a team of two domain experts and myself as Scrum coach to put together the RFP. We decided to use Scrum to create the RFP. During “Sprint Zero”, which only lasted a week, we were briefed by the product owner. Then we created and estimated the product backlog. We reserved a meeting room for the duration of the project, which gave us a place for the burn down charts, task board, and other information that gets hung on the wall in a Scrum project.

We considered the RFP itself to be a product, so we used the agile workshops originally defined by Mike Cohn [3] to figure out who our readers are and what they want from the document. The users included:

- Our own management, who need a clear vision of what the product will be and what it means for the company
- Prospective suppliers, who need to understand the business process which shall be automated and what the new system should do.
- “Everybody” who needs to understand how Scrum would be applied in this project.

We came up with a table of contents which included important elements of a classical Project Plan [4]

1. Project Charter
    a) What are overriding the corporate goals -- taken from the corporate website
    b) Elevator Pitch - what will the product do for its users?
    c) Product Mission - what will the product do for the company?
2. Current Situation: How is the problem being solved today?
3. Desired Situation: How should this system solve problem?
4. Risk Analysis: What has gone wrong in the past and how do we want to prevent that in the future. What are the big non-functional problems that have to be solved?
5. Project Management Procedures, Roles and Responsibilities: A description of Scrum and its roles.
6. Budget Recommendation: How much should the project allowed to cost based on the benefit it will bring to the company?

Each entry in the table of contents became a theme for grouping the stories. Each story corresponded to some content in the RFP or an appendix. For example, we had stories like ‘Current Process Diagram’ or ‘Project Management Procedures’ which corresponded directly to (sub-) chapters in the RFP.

Sprint Planning and Demos

Each sprint lasted 2 weeks. The Sprint Review and Planning Meetings were not as formally separated as in Scrum by the book. The Product Owner gave us guidance, first in the Kick-Off meeting, later during the Sprint Reviews and the team decided how to translate that guidance into content in the RFP. So the team was truly self organizing and assumed some of the duties of the Product Owner.

Based on our initial estimates, we set the following themes for each Sprint.

1. Project Vision
2. Current State
3. User Interviews
4. Desired State
5. Final Document
6. Retrospective/Next Steps

It didn’t quite work out that way. In the Sprint 1 Retrospective, we realized that we should be delivering the RFP incrementally and that we should work toward a definition of done. So we decided that a Story is only finished when the corresponding information has been integrated into the RFP document and when that chapter has been reviewed within the team for content, spelling and grammar, and the updated RFP is posted on the Wiki. After Sprint 3, we could have given the RFP to potential suppliers after any Sprint (perhaps with a little formatting), and they could have starting the bidding the process.

Sometimes we did break stories down ‘horizontally’, e.g. each user interview was a story, even though a user interview produced no content for the RFP. Such stories had to produce a clearly defined deliverable which was useful for creating the final deliverable. So a user interview story produced a summary of the interview which the team used a basis to discuss the feedback.

The First Sprint

The application should automate the information flow between the company, its customers and its suppliers. The process is very complicated and had been analyzed previously, so we wanted to build any work which was already available.

The first sprint focused on two issues:

1. What is the product charter: Who is the product for? What should it do for them? What should it do for the company?
2. What previous analysis was available?

The first drafts of the Product Mission and Elevator Pitch provoked a lot discussion with the product owner and beyond. Small differences in the wording of either can have tremendous impact on the product, adding or removing many person-months of work. Lack of clarity increases the risk of building the wrong product, not including critical functionality, wasting resources on unneeded functionality or endlessly retargeting the product as management changes its mind about what the vision is.

Using Scrum for the RFP Project

We used classic Scrum to create the RFP. Stories corresponded to chapters or subchapters of the RFP. The stories were estimated in Story Points. At the beginning / end of each sprint we discussed the results with the product owner and planned the “functionality” for the next sprint. Each story in the sprint backlog was broken down into tasks and estimated in hours. We used release and sprint burn down charts to track our progress, occasionally adjusting the scope to ensure that the project would deliver the RFP on time.

We used ‘tangible tools’ - cards and flip charts - to manage the backlog, tasks and burn down charts.

By the end of Sprint 3, we had a document in close to final form. There is a special moment of recognition in the Product Owner’s eyes, when he sees that he will get his product as wants it. People who have worked on a waterfall basis aren’t used to experiencing that until much later in the project, if at all. By the last sprint, we were starting to wonder if the additional work we were investing wasn’t just ‘gold plating’, which made it easier to declare the RFP ‘Done!’

Contents of an Agile RFP

Agile projects set different priorities than traditional approaches and this has an impact on the contents of the RFP. The most important differences between agile and traditional approaches can be summarized as follows:

	Agile Project	**Traditional**
Objective of Project	Meet goals and needs of Organization, Users and Customers by providing them functionality which helps them accomplish their goals.	Realize defined scope within defined time and budget
Scope	Negotiable - realize the minimum set of features required to meet the objectives of users and customers	Precisely defined in advance. Often expressed as wish list, which gets cut down in the process of negotiations to meet budgetary goals.
Scope Changes	Embraced. The priority of not yet implemented functions can be downgraded in favor of new requests which are judged more important.	Discouraged. Changes are often a source of delay or cost overruns. Change requests are often used by vendors to restore profitability to projects where fierce competition resulted in unprofitable basic prices.
Time	Release quickly to start generating ROI	Release once with all functionality
Quality	Actively defined and agreed upon through definition of ‘done’, which is confirmed at the end of every Sprint	Assured in a separate QA-Phase. This increases the risk of delay and cost overruns, because errors detected late are more expensive to find and fix.
Trust	Pre-Requisite for working agilely, difficult to establish before work starts.	Attempt to compensate lack of trust through contractual process, penalties, etc.
Cost	Ideally planned proactively as a function of the value which the project should produce.	
Risk	ROI is managed continuously by the product owner. Incremental delivery, regular inspections and prioritization (delivering the most valuable work first) are the primary tools for mitigating risk.	Risk of cost overrun is a major factor in planning the project. Fixed Price/fixed scope, cost ceilings, penalties for late delivery are used to minimize financial risk

Because of the differences, our RFP had a similar structure but different contents than a typical, waterfall oriented RFP:

1. Introduction (the Company introduces itself)
2. Goals
    a) Corporate Goals
    b) System Goals (Product Mission)
    c) Project Goals (Elevator Pitch)
3. Budget Considerations
    a) Potential Benefits and Savings
    b) Investment Recommendation
4. Process (Scrum)
    a) Risk Management
    b) Project Organization
    c) Scrum Roles and Responsibilities
    d) Scrum Work Flow
    e) Quality Management
5. Current Situation
6. Desired State
    a) Personae (Roles)
    b) User-Stories
    c) Workflows
7. Selection Process
    a) Trial Run / Competitive Sprints

Introduction and Goals

Agile is goal and business value oriented. So the first priority of the RFP is to communicate the goals of the company, the project, the product and its users and other stakeholders. The functionality of the system is a means to the end. So the RFP started with a presentation of the company, its goals and the marketing and functional goals of the system.

This section was the subject of substantial discussion with management. Who are the intended users? How is the market changing and how should this product be positioned to react to those changes? What should the product do and what should not do? The results were realistic expectations on product and a clear vision of what should be built and why.

Budget Considerations

For agile projects, a relationship of trust between customer and supplier is essential. Trust means that each partner does not need to fear that his inadequacies will be exploited by his partners. Trust enables conflict at the level of ideas - What is best for the product, for the customer and for the customer’s customers? With trust, win-win situations are possible. With trust, it is possible to give in and compromise. Establishing trust requires openness and honesty and takes time (together) to build and develop. This encouraged us to include the budget considerations directly in the RFP and presently clearly the expected risks of the project, even though it meant exposing our own warts.

It was difficult to come up with a believable model for the benefits (costs savings, new revenue, new business models etc.) of the system. There so are many consultants with spreadsheets of dubious value. So we chose to simply frame the problem. How many employees will be affected by the system? How much more business could they process if their productivity were raised by 5%, 10%, etc. Or how much money could be saved?

There was not one answer to this question, but a range of values for discussion. For each of these ranges, we calculated a value based investment recommendation, based on the double worst case analysis. [5]

Process (Scrum)

The risk analysis focused primarily on actual problems. We had done a retrospective on previous projects and had a good idea on what we wanted to improve or prevent, e.g. SW reliability problems (internal quality), fitness-for-use problems (external quality), project management and oversight problems, supplier dependencies, and misalignment of short term vs. long term budget optimizations.

Many of these issues can be addressed effectively by using Scrum, so we defined Scrum as the process for software development and for escalation management. We included a description of the Roles, Rituals and Artifacts. We included some additional metrics on Software Quality, Acceptance Tests defined and passed, Unit Tests defined and passed, which should be presented at every sprint demo.

Escalations shall be handled by a “Management Oversight Committee” modeled on Ken Schwaber’s Enterprise Transition Team. Their job is to resolve impediments that the ScrumMaster can’t.

Originally envisioned as two separate chapters, the Risk Analysis chapter became the introduction to the process chapter and the justification for using Scrum.

Some issues aren’t directly addressed by Scrum. How do you ensure system maintenance over a 10 year period? This is both a financial question and an engineering question. We simply raised these issues with the suppliers in the RFP.

Current and Desired Situation

Next, an understanding of the context is essential to make good decisions about how to implement the product. The current situation described the process as it existed. This made clear how the business functioned and helped identify optimization potential, both in the execution process and more importantly, in the customer’s decision process, which could lead to more business for the company.

The section ‘Desired State’ reflected most clearly the difference inherent in an agile approach. Rather than focusing on the functionality of the system, this section focused on the users (roles or personae), their goals and what they want to accomplish with the system.

We spent the most time analyzing the current and desired situations. As when planning the RFP Project, we started out by brainstorming to identify the user roles ("Personae") and their function in the process. Some of the roles were obvious: production workers for our company, our customer and our suppliers. As the purpose of this system is to reduce operating costs, these were clearly the people who would most intensively use the system.

An important discovery in the Personae workshop was that the primary (full time) users of the system are not the purchasers or influencers. The users' managers, the suppliers' and customers' sales, marketing and management and even our own top management will be 'only' occasional users, but have much more to say about the success of the system than the primary users. By giving these users real value, we create Exciter Functions [6] for those users who are most essential to success of the product.

Further, support users and the system developers also needed functionality to do their jobs, and creating the right functions (including delegating support functions back to the user) could lower operating costs substantially.

This quickly produced a draft of the desired situation, not so much in terms of the detailed functionality, but of the user roles and how they will change with the new system. It also showed how the system could change the relationships with customers and suppliers, open new business opportunities or generate new revenue from existing customers.

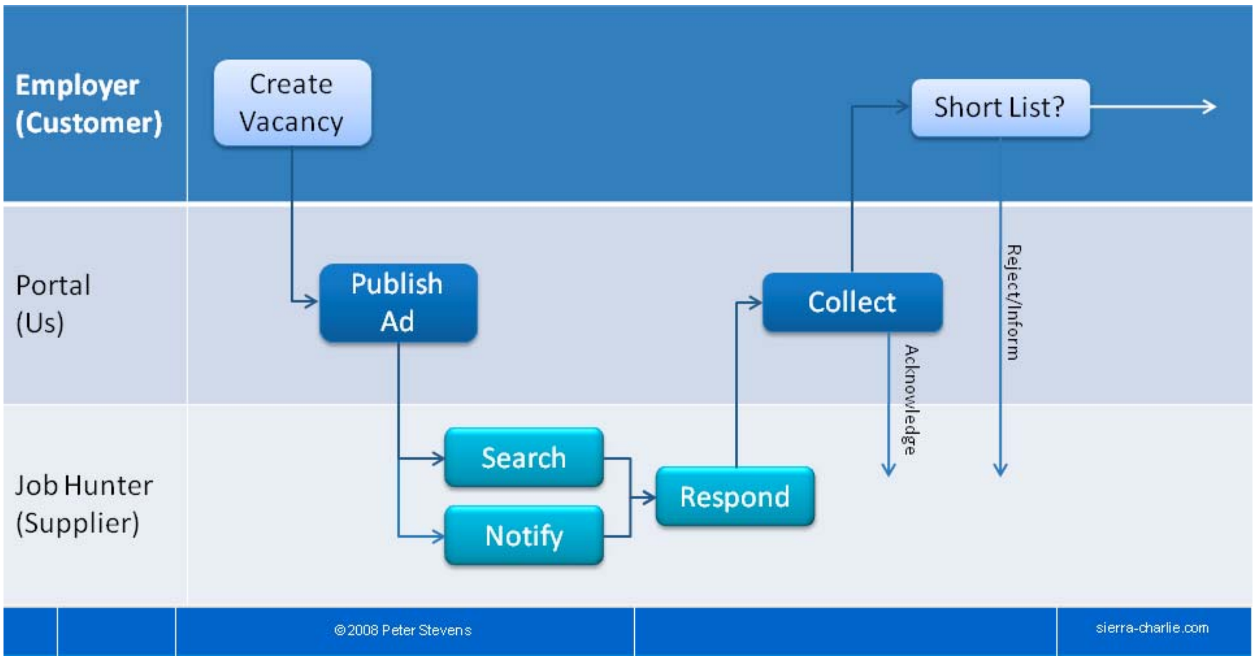
Specify the Product as User Stories

We ended up writing a complete set of user stories for all major users of the system. The result was 20 pages long, so we moved them into an appendix. In the main document, we just included a top level description of each user role, his/her duties and what that role expects from the system.

Augment the Description with a 4 Track Value Flow Chart

We had set out to describe the processes using Value Steam Mappings [7]. After all, we wanted to eliminate waste. However, the times involved in the processes were so variable that it was not possible to come up with numbers which would be generally accepted. Instead, we created a 4 Track Value Flow Chart. Each track corresponds to one of the primary actors:

- End Customer (Our customer's customer)
- Our Customer
- Us
- Our Supplier



Sample Three Track Value Flow Chart for a fictional job portal.

The tracks are laid one on top of the other and the horizontal axis represented time. The flow chart shows activities, processes and decisions, and who does each. The flow analysis makes it clear where productivity improvements could be made, when key decisions are made and by whom, and what interfaces between companies are present and necessary.

Selection Process

How do you know if a team can do agile? How do you know that they can work with you effectively? Which team is really the better choice? This is a hard question to answer. The classical approach suggests that the vendor should produce extensive documentation about the proposed solution (at no cost to the customer) and limit the risk through fixed price/fixed scope contracts, cost ceilings, bonus/penalty incentives etc.

These approaches do not work well with agile development projects, because they create competitive games between vendor and customer, undermine the trust which is necessary to work together effectively, and are challenged reacting to variability in scope.

Competitive Sprints, an agile selection process

Some people would like to believe that building complex software is like going to the grocery store: pick a candy bar off of the shelf, ask what it costs and decide to buy it. You get no risk and quick gratification. But building custom software is more like building a race car. A special one-off product to meet exactly the needs of its sponsor: win races.

As a customer, what are you really buying when you contract for software development? You may think you are getting a solution. But what you are really getting is an implementation team. And risk is always part of the bargain. So what you really want is a team you can trust to build your product and to minimize the risk of that choice.

Competitive sprints are a lightweight, lean approach to selecting a software development partner which should dramatically reduce the risk and cost to all parties.

Selecting a vendor through a bidding process is expensive and risky

The process of writing a huge RFP, evaluating and eventually accepting complicated bids from largely unknown vendors is wasteful, risky and expensive. A customer will often invest substantial effort into creating a Request for Proposal (RFP). I've seen RFPs whose size is measured in binders and the effort to produce them measured in man-years. A vendor will often invest comparable effort into winning a big project. On both sides, this effort produces mostly paper. These artifacts have no value except as means to the goal of selecting a vendor.

On the vendor side, this effort has to be amortized during the execution of the project itself. As there is only one winner, the others make a substantial effort and earn no money. So this risk gets passed on in the form of higher prices on successful bids.

Competitive bidding can mean offering ruinous prices. When “winning” means producing at a loss (sometimes to the point where the supplier goes out of business [8] [Automatic English translation 9]), even the customer loses. The supplier may have no choice but to play the change-request game to achieve profitability.

A Lean Approach to reduce cost and risk for all parties

How do you maximize trust and minimize risk and cost? Once trust has been established, customers are often willing to work with agile companies on a time and materials basis. Trust greatly reduces administrative overhead and enables a collaborative approach, so establishing trust should be the first priority in a project.

Lean thinking encourages us to see the whole, eliminate waste and defer decisions until enough information is available to make that decision with reasonable risk. Deferring potentially expensive decisions until their implications are clear is usually advantageous.

Simply asking for quotes and having a few talks with the sales & consulting staff is just not enough to establish trust or clarify all the open questions. Working together with the team and evaluating real results would be much more effective.

To select a vendor, after creating an agile RFP as described above:

1. Prequalify potential vendors
   a) Identify potential vendors and send them the RFP.
   b) Ask the vendors questions, in particular about their experience with agile methods, which will allow you to quickly eliminate uninteresting vendors.
2. Invite the survivors to bid on the project
3. Select the final short list of 2 vendors who will be invited to start work on the project.
4. Both vendors work on the project in parallel until a clear favorite is established. Both vendors get paid and both are expected to produce increments of working software according to the rules of Scrum.
5. After a few sprints, select one vendor to finish the project.

By deferring the final decision until you have actual experience with the candidates, you reduce the likelihood of picking a candidate that “looks good on paper” but cannot really deliver software.

How to prequalify vendors

Identify your candidates and send them the RFP. Ask them questions which will separate the wheat from the chaff, for example:

- Please present the team which will carry out the project. How much experience do they have with Scrum and XP (Extreme Programming)?
- Are you willing to organize the project according to Scrum (as described in the Process chapter)? What experience do you have with agile projects on this scale?
- Please estimate the user stories in story points. What is your estimate of overall complexity (size) of the system in Story Points?
- What is your expected team velocity in Story Points per Sprint?
- Given our target budget, how much of the functionality do you think can be realized?
- Given the ground rules, are you willing to participate in the competition to select the final vendor?

If the vendors are used to working on an agile basis, they will have no problems with these questions. If they are not, they will probably not even be able to respond, especially if deadlines are kept tight.

You will need to meet with prospective teams for a day or so to answer their questions about the user stories. Afterwards the vendors should be able to size the system and answer your questions quickly.

If there are more than two vendors still in the running, you will need to use the answers and the results of the interviews to trim the field down to two vendors, who then participate in the competition.

Hedge your bets on productivity differences between teams

Productivity differences between individual developers can be a factor of 10 to 20. Teams converting to Scrum often report a 3 fold increase in productivity within 6 months. The best Scrum teams have reported improvements of a factor of 10 compare to industry averages. The difference might be technical ability, but human chemistry issues are just as important, if not more so.

Even if the difference between to the top two contenders is only 25%, investing 10 to 20% of the development budget to hedge your bets reduces your risk substantially and may pay off dramatically.

Let us assume that you plan to spend $2.4 Million over 12 months, or $200'000 / Month to develop the software. If you add one month's effort to the budget, that would raise the total by 8%. But given the productivity differences between teams, even investing an additional 25% probably yields a positive ROI. Furthermore, the cost of delay while you take three months to pick a partner without producing any usable software should be much larger than the cost of redundant development for a short period.

How to run the contest

Here are the ground rules. There are two vendors. Both are going to start your project according to the process you defined in the RFP and continue for a defined trial period (probably one to three months, but not more than 25% of the total project duration). After each sprint, both players present an increment of working functionality and you will decide which partner you want to continue working with. The winner gets full compensation for the initial sprints and a contract for the rest of the project. The loser also gets paid (perhaps only 50 or 75%) and does not get a contract.

Here are the steps in the competition.

1. Agree on other playing rules: who are the team members? May staff participate who are not invoiced? Is overtime allowed? Who owns the software and ideas which are produced by the loser? You may need to ensure that quality does not get sacrificed for quantity as you will be producing code which one day will go live.
2. Agree on the definition of done. This probably should include points which confirm that the partner is capable of test driven development and continuous integration. The teams should not incur technical debt. The same definition should apply to both contestants.
3. Prioritize the backlog and, working with both vendors, create a set of fine grained stories which should be implemented during the trial period.
4. Hold the first sprint planning meeting together with both contestants, so both teams get the same initial briefing. Both teams get a defined period to implement their increments of functionality.
5. If the trial period is longer than one sprint, then continue with the Scrum process until the trial period is over. Each team should work independently.
6. Finish the / each Sprint with a Sprint Review and Retrospective. This should only include the implementation team and the product owner, as it is part of the Scrum process. When the last sprint retrospective is complete, the competition is complete.

If the teams want to give a sales demonstration, this should happen after the competitive sprints are finished.

At the end of the trial period, you have not just qualitative and quantitative data on which to base a decision. You have experience working with your new partner. You have some working software (or not). In short, you have much more information to base your decision on.
This approach does present some challenges. Managing one team can be very demanding for the product owner. Managing two teams might be difficult. This will be even more dramatic if the teams do more than one sprint, as the sprint and product backlogs will surely diverge quickly.

Who wins?

One the one hand, everybody wins.

The risks and up-front costs of producing and responding to the Agile RFP are much lower than the traditional approach. By working with the teams, you build confidence that you can work with them and that they can build the software you need. You can judge the teams based on working software rather than attractive presentations or other artifacts.

By starting to work on a solution early, you reduce your time to market. You will probably get a better solution, because the competition will spur everyone's creative juices. Even the loser will have some good ideas which will improve the final result.

On the other hand, you actually have to choose a vendor. On what criteria should you decide? Just picking the one who develops the most functionality is risky. It encourages the developers to incur technical debt to show more exciting functionality. Economic criteria will still play a role. Whatever you choose, they should ensure that you pick a vendor with whom you want to work and in whom you have confidence that they can deliver the solution you required.

Retrospective

Scrum worked well to manage the creation of the RFP. We defined the scope and met the essential objectives of the project -- even if some of the stories did not get implemented. The RFP served its purpose by focusing on the goals of its users. A definition of done was very helpful to focus on creating the document on a chapter by chapter basis. However the product owner did not really do incremental “acceptance testing”, so there were a lot a changes and the end of the project.

The customer decided to keep the implementation in house, rather than contract it out, so the competitive sprint approach was not tested.

Three people worked on the RFP at approximately 50%. Approximately 4 1/2 Person-Months were invested in the RFP. Was this too much? This is the big unanswered question. In an agile process, the objective is to create the specification just in time through direct communication between team and product owner. This is not possible before a supplier has been selected.

We did learn a lot about the desired application and shifted the center of gravity of the project in ways we would not have expected had we not done the user centered design. By writing the stories ourselves, we remained business focused and not technology focused. We have a complete product backlog, but have not thought much about release strategies (time to market) because without an implementation team, we had no way to estimate the stories.

But should we really have written 20 pages of user stories? I am not sure. I had the feeling we were over-specifying. The more we produce, the more we will have to communicate to the implementation team. The more the team gets told what to do, the less they will think about the problem themselves.

Future work

This approach is a step away from a classical submission process, but is not yet purely agile. Some thoughts for next steps:

1. Repeat this process in another context. Our process had no aspirations to creating a generalized process. So your RFP might and probably look different than this project’s results.
2. Use competitive sprints in an actual bidding process
3. Is it possible to involve potential vendors already in the RFP creation project? At least theoretically, it would be better to have one or more people from the implementation team involved from the beginning. Whether consultants from competing companies would have an incentive to work together is interesting question.

References

1. http://tinyurl.com/cnog6b
2. http://groups.yahoo.com/group/scrumdevelopment/message/30679
3. http://agilesoftwaredevelopment.com/blog/peterstev/creating-scrum-product-backlog-start-users
4. Inspired by http://blogs.techrepublic.com.com/tech-manager/?p=581&tag=nl.e053
5. http://agilesoftwaredevelopment.com/blog/peterstev/big-development-project-how-much-does-it-cost
6. http://agilesoftwaredevelopment.com/blog/peterstev/filling-product-backlog-go-excitement
7. M. & T. Poppendieck, Implementing Lean Software Development, pg. 83-92
8. http://www.vbz.ch/vbz_opencms/opencms/vbz/deutsch/DieVBZ/TramBus/VBZ-FahrzeugeImEinsatz/fahrzeuge/DasCobra-Tram/entwicklungsgeschichte.html
9. http://translate.google.ch/translate?u=http://www.vbz.ch/vbz_opencms/opencms/vbz/deutsch/DieVBZ/TramBus/VBZ-FahrzeugeImEinsatz/fahrzeuge/DasCobra-Tram/entwicklungsgeschichte.html&sl=de&tl=en&hl=de&ie=UTF-8

Based on a series of articles originally published on AgileSoftwareDevelopment.com.

Database Locking: What it is, Why it Matters and What to do About it

Justin Callison, Principal Consultant, justin @ peakperformancetechnologies.com
Peak Performance Technologies Inc, http://www.peakperformancetechnologies.com/

Introduction

"Know your enemy and know yourself and you can fight a hundred battles without disaster." Sun Tzu (The Art of War)

Database locking is a varied, evolving, complicated, and technical topic. As testers, we often think that it belongs in the realm of the developer and the DBA (i.e. not my problem). But to both functional and performance testers, it is the enemy and has led to many disasters (as the presenter can personally attest).

However, there is hope. This paper will shed light on the nature of database locking and how it varies between different platforms. It will also discuss the types of application issues that can arise related as a result. We will then look at ways to ferret out these issues and to resolve them before they sneak out the door with your finished product. Armed with this understanding of the enemy and how it relates to your application, you'll be much better able to avoid disaster.

1. Scope

The breadth and depth of this topic necessitates that scope be constrained. The scope of this paper has been chosen with the following considerations in mind:

- The audience is QA professionals
- The audience does not have significant database experience
- Core concepts can be generalized once understood

Specifically, the scope will be constrained to:

- "Transactional locking" (not all types of locking)
- Oracle and SQL Server

2. Why do Databases Lock?

So why does locking occur in a database? As in other systems, database locks serve to protect shared resources or objects. These protected resources could be:

- Tables
- Data Rows
- Data blocks
- Cached Items
- Connections
- Entire Systems

There are also many types of locks that can occur such shared locks, exclusive locks, transaction locks, DML locks, and backup-recovery locks. However, this paper will focus on one specific type of locking that I will call “transactional locking”.

2.1 ACID Properties of Transactions

Most of what we’re calling transactional locking relates to the ability of a database management system (DBMS) to ensure reliable transactions that adhere to these ACID properties. ACID is an acronym that stands for *Atomicity*, *Consistency*, *Isolation*, and *Durability*. Each of these properties is described in more detail below. However, all of these properties are related and must be considered together. They are more like different views of the same object than independent things.

2.1.1 Atomicity

Atomicity means all or nothing. Transactions often contain multiple separate actions. For example, a transaction may insert data into one table, delete from another table, and update a third table. Atomicity ensures that either all of these actions occur or none at all.

2.1.2 Consistency

Consistency means that transactions always take the database from one consistent state to another. So, if a transaction violates the databases consistency rules, then the entire transaction will be rolled back.

2.1.3 Isolation

Isolation means that concurrent transactions, and the changes made within them, are not visible to each other until they complete. This avoids many problems, including those that could lead to violation of other properties. The implementation of isolation is quite different in different DBMS’. This is also the property most often related to locking problems.

2.1.4 Durability

Durability means that committed transactions will not be lost, even in the event of abnormal termination. That is, once a user or program has been notified that a transaction was committed, they can be certain that the data will not be lost.

3. Examples of Simple Locking

The example below illustrates the most common and logical form of transactional locking. In this case, we have 3 transactions that are all attempting to make changes to a single row in Table A. U1 obtains an exclusive lock on this table when issuing the first update statement. Subsequently, U2 attempts to update the same row and is blocked by U1’s lock. U3 also attempts to manipulate this same row, this time with a delete statement, and that is also blocked by U1’s lock.

When U1 commits its transaction, it releases the lock and U2’s update statement is allowed to complete. In the process, U2 obtains an exclusive lock and U3 continues to block. Only when U2’s transaction is rolled back does the U3’s delete statement complete.

This example shows how the DBMS is maintaining consistency and isolation.

Time	User 1 Actions	User 2 Actions	User 3 Actions
1	Starts Transaction		
2		Starts Transaction	
3			Starts Transaction
4	Updates row 2 in table A		
5		Attempts to update row 2 in table A	
		U2 Is Blocked by U1	
6			Attempts to delete row 2 in table A
			U3 Is Blocked by U1
7	Commits transaction		
		Update completes	*U3 Is Blocked by U2*
8		Rolls back transaction	
			Delete completes
9			Commits transaction

4. Locks Outlive Statements

It is critical to understand that locks will often remain after a statement has finished executing. That is, a transaction may be busy with different, subsequent activity but still hold locks on a table due to an earlier statement. Transactions may even be idle. This is especially dangerous if the application allows user think time within database transactions.

This concept that locks outlive statements may seem obvious, but it is often forgotten when considering the impact of locking, so it is very important to remember.

5. Issues that can occur

So what does this mean for your application? Well, there are a number of problems that can be caused by database locking. They can generally be broken down into 4 categories: Lock Contention, Long Term Blocking, Database Deadlocks, and System Deadlocks. Each of these types of issues is discussed in more detail below, including examples.

Note: The examples used are quite extreme, but are meant to be illustrative.

5.1 Lock Contention

Lock contention occurs when many database sessions all require frequent access to the same lock. This is also often called a “hot lock”. The locks in question are only held for a short time by each accessing session, then released. This creates a “single lane bridge” situation. Problems are not noticeable when traffic is low (i.e. non-concurrent or low-concurrency situations). However, as traffic (i.e. concurrency) increases, a bottleneck is created.

Overall, Lock Contention problems have a relatively low impact. They manifest themselves by impacting and limiting scalability. As concurrency increases, system throughput does not increase and may even degrade (as shown in Figure 1 below). Lock contention may also lead to high CPU usage on the database server.

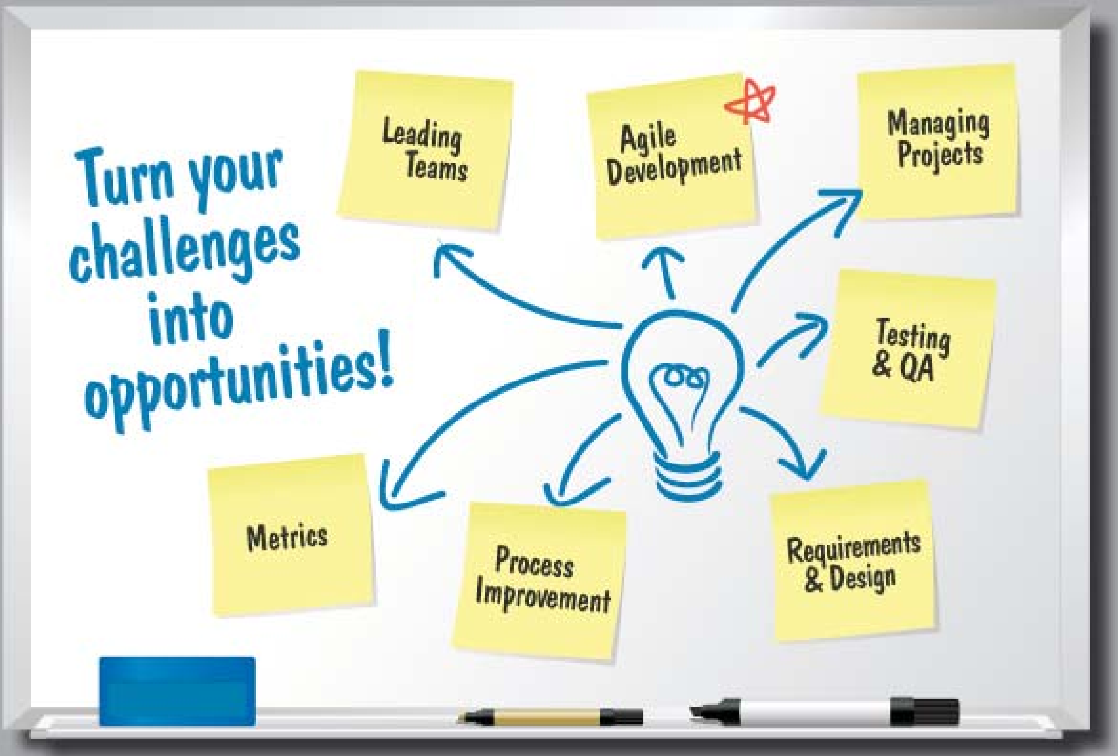
Advertisement - Better Software Conference - Click on ad to reach advertiser web site

BETTER SOFTWARE
CONFERENCE & EXPO

JUNE 8–12, 2009
LAS VEGAS, NV
THE VENETIAN

Conference Sponsor: RALLY SOFTWARE

www.sqe.com/go?BSCE09M&Tad



KEYNOTES
BY INTERNATIONAL EXPERTS

Tim Lister
Atlantic Systems Guild, Inc.

Andy Kaufman
Institute for Leadership Excellence & Development, Inc.

Michele Sliger
Sliger Consulting, Inc.

Jonathan Kohl
Kohl Concepts, Inc.

Agile LEADERSHIP SUMMIT by APLN

NEW FOR 2009!
a co-located event

Agile Leadership Summit
Friday, June 12, 2009

SOFTWARE QUALITY ENGINEERING
CONFERENCES
www.sqe.com

TUTORIALS
WORKSHOPS
CLASSES

The best way to identify a lock contention problem is through analysis of statistical information on locks provided by the DBMS (see monitoring section below).

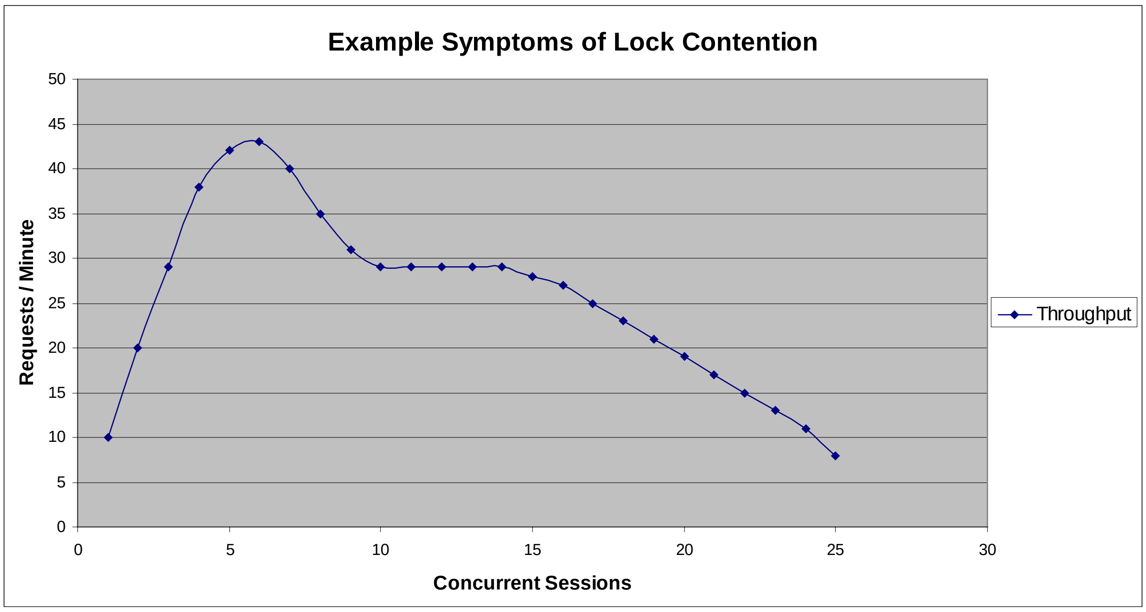


Figure 1: Example Symptoms of Lock Contention – Scalability

5.1.1 Example: Statistics Table

A web application has a requirement to record the number of times each page is loaded. This has been implemented with a database table where the access counts for each page are stored, with one row for each page. Each time that a request is made to a page, the appropriate row in the tables is updated to increase the request count number by 1.

The related update statement is not a problem for low concurrency situations, as the requests are fast and the same page is not loaded concurrently. However, as load increases, popular pages are concurrently being accessed. This leads to contention and limits scalability.

5.1.2 Example: Order ID Storage

An order processing system requires that Order IDs be unique and sequential. This is implemented through use of a database table to store the value of the next order ID. As each order is processed, the transaction gets the value for the next order ID then updates it with the next ID.

Under low concurrency circumstances, order id generation does not have a significant impact. However, as the number of concurrent orders increases, a bottleneck is created due to contention on the associated lock, limiting order throughput.

5.2 Long Term Blocking

Long Term Blocking is similar to Lock Contention in that it involves an object or lock that is frequently accessed by a large number of database sessions. Where it differs is that in this case, one session does not release the lock immediately. Instead, the lock is held for a long period of time and while that lock is held, all dependent sessions will be blocked.

Long Term Blocking tends to be a much bigger problem than Lock Contention. It can bring an entire area of functionality or even a whole system to a stand still. The locks involved in these scenarios may not be "hot" enough to lead to Lock Contention problems under normal circumstances. As such, these problems may be intermittent and very dependent on certain coincidental activity. These are the most likely to lead to "disasters" in production due to the combination of devastating impact and difficulty to reproduce.

The consequences of long term blocking problems may be abandonment. However, these problems can also often lead to further problems as frustrated users re-submit their requests. This can compound and exacerbate the problem by leading to a larger queue and consuming additional resource. In this way, the impact can expand to consume an entire system.

5.2.1 Example: Order ID Batch Processing

Consider the same system described in 5.1.2 above. But now imagine that the system also has a batch process that is set to run at 7 p.m. This batch process is optimized to complete multiple orders within one transaction. As such, it will acquire a lock on the Order ID row at the beginning of processing. If it then takes 15 minutes to process the entire batch, that lock will be held for the duration and no other orders can be processed until it completes, commits the transaction, and releases the lock. This situation is illustrated in the graph below, where order processing times are very fast (a few seconds) during most of the day, but then shoot up to 900 seconds during the 7 p.m. hour.

Users trying to process orders during this time think that the system has gone down. Many resubmit their requests, leading to a queuing of duplicate requests which cause problems later due to duplicate orders. The repeated requests also lead to exhaustion of threads and database connections, making the system unable to respond to any requests, even those unrelated to orders.

This might not have been found during testing, because only small batches were tested or batch processing was not tested concurrently with regular usage.

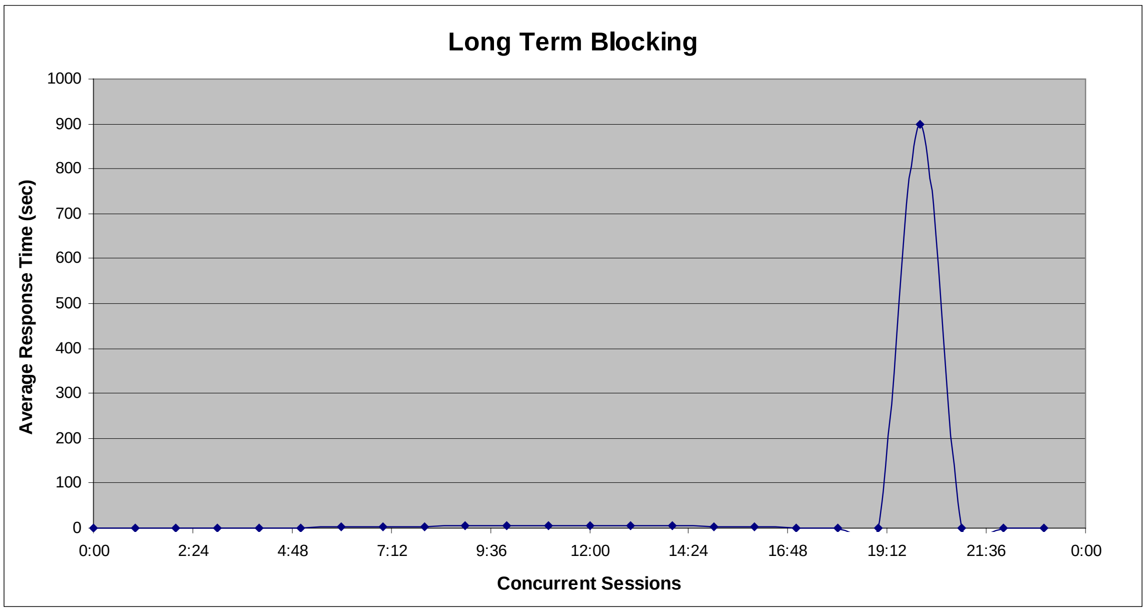


Figure 2: Example Symptoms of Batch Locking

5.3 Database Deadlocks

Database Deadlocks occur when 2 or more transactions hold dependent locks and neither can continue until the other releases. Below is a simple illustration of a deadlock.

Time	User 1 Actions	User 2 Actions
1	Starts Transaction	
2		Starts Transaction
3	Updates row 2 in table A	
4		Updates row 10 in table B
5	Attempts to update row 10 in table B	
	U1 Is Blocked by U2	
6		Attempts to update row 2 in table A
	U1 Is Blocked by U2	*U2 Is Blocked by U1*
	DEADLOCK!!!	

This example is the simplest type. Deadlocks can be much more complicated, involving different types of locks, and involving more than 2 sessions. The good news is that the DBMS will recognize these situations and resolve them. The DBMS will choose one “victim” and roll back their transaction. The DBMS tends to choose the “victim” by determining which transaction will be easiest to roll back. However, unless the application is designed to react to deadlock errors and retry, there will be a negative end-user impact.

Deadlocks generally lead to intermittent errors. They are *almost* always caused by application logic problems. These generally occur when the application does not access data in a consistent sequence. They are also very timing and data dependent, which can make them very hard to reproduce.

Note: Deadlocks can occur within other components as well, such a JVM or the CLR.

5.4 System Deadlocks

System Deadlocks (aka Distributed Deadlocks) are similar to a Database Deadlock, in that they involve dependent locks. However, they involve locks that are outside the database (e.g. in the JVM) as well. As a consequence, the database cannot identify the deadlock situation. It simply appears to be a normal blocking situation. The same will be true for the external component. As such, these situations are usually not resolved until a timeout occurs or one of the components is restarted.

System deadlocks are relatively rare, but they do occur. And when they do, they can lead to system outages as well. In many cases, the impact will grow because the system deadlock will lead to secondary long term holding of locks which result in the Long Term Blocking for other sessions.

5.4.1 Example: Start-up Deadlock

A J2EE application has a start-up thread that’s responsible for initializing various parts of the system. A “system change number” is stored in a database table and is updated every time that key tables are changed. The first start-up task makes such a change and updates the system change number, acquiring a lock as it does.

Subsequently, the application executes an EJB method that has been configured with the “Requires New” property. As such, the application server must create a new database connection and transaction to execute that EJB method. This particular method also leads to changes that require the same system change number to be updated.

At this point the start-up thread is waiting for the EJB method to complete. However, this method is being blocked by the database lock associated with the first start-up transaction. The database sees a simple blocking situation, as does the JVM. However, this problem will never resolve and the system never starts up.

5.4.2 Example: Session Counts

A clustered .NET web application has requirements to track the number of sessions logged into each application server and the entire system. Each application server tracks the session count with in memory counters while the system wide total is tracked in a database table. The in-memory session tracking was protected with a locking mechanism such that three methods must be called in order to update it. The update of the database is completed with a single method. When implementing the login, the developers coded the following sequence:

1. aquireSessionCountLock
2. updateSessionCountValue
3. updateDBSessionCount
4. releaseSessionCountLock

This was done so that, if the database updateDBSessionCount call failed, then the updateSessionCountValue call could be reverted to ensure that a discrepancy did not occur between the two session counts.

However, another developer implemented the logout request in the following way:

1. updateDBSessionCount
2. aquireSessionCountLock
3. updateSessionCountValue
4. releaseSessionCountLock

That is, the database will be updated, after which the in-memory counter is updated (including acquisition and release of locks). This would not be a problem under low concurrency situations. However, with sufficient load and concurrency, the following situation would occur, creating a System Deadlock.

Time	User 1 Actions	User 2 Actions
1	Login Request Received	
2		Logout Request Received
3	Executes aquireSessionCountLock	
4		Executes updateDBSessionCount
5	Executes updateDBSessionCount	
	U1 Is Blocked by U2's DB Lock	
6		Executes aquireSessionCountLock
	U1 Is Blocked by U2's DB Lock	*U2 Is Blocked by U1's App Lock*
	SYSTEM DEADLOCK!!!	

Neither the CLR nor the DB would be able to detect that this deadlock and it would go unresolved. Subsequent Login requests to this app server would be blocked in the aquireSessionCountLock call. Requests to other application servers would be blocked in the updateDBSessionCount call due to the database lock. As a result, the system would become non-functional.

6. Complications and Platform Variation

The situations discussed so far have involved database locking that is quite simple, logical, and expected. Most problems can be avoided with sound development practices. However, there are further complications that can lead to additional types of locking that result in more frequent locking and locking that may seem illogical or not strictly necessary. These complications are also generally a result of DBMS differences. When the frequency of locks and locking events increases, the frequency of the problems described above also increases. And they also begin to appear in unexpected situations.

6.1 Select Blocking

Select blocking is closely related to the isolation property of ACID transactions. In order to ensure that transactions are isolated, the DBMS must be sure that changes made by one transaction are not visible to others until the transaction commits. For example, if one transaction updates a row in a table and another session tries to select the updated row, the DBMS cannot allow the changes to be seen by the second session.

The solution to this problem is variable. Oracle's implementation is to maintain separate versions of the related data blocks. The uncommitted changes made in one transaction are visible within the transaction that made them, while other sessions will see the old, unchanged version. In this way, isolation is maintained and *select statements never block.*

SQL Server and other DBMS' have traditionally implemented this differently. Instead of maintaining separate versions, they use locks to protect access to the changed data. This leads to *Select Blocking*. The difference is illustrated in the examples below.

Time	User 1 Actions	User 2 Actions
1	Starts Transaction	
2		Starts Transaction
3	Updates row 2 in table A	
4		Attempts to read row 2 in table A
		Select statement completes, returning data unchanged by U1
5	Commits transaction	
6		Attempts to read row 2 in table A
		Select statement completes, returning changed data

Table 1: Select Behaviour in Oracle

Time	User 1 Actions	User 2 Actions
1	Starts Transaction	
2		Starts Transaction
3	Updates row 2 in table A	
4		Attempts to read row 2 in table A
		U2 Is Blocked by U1
5	Commits transaction	
		Select statement completes, returning changed data

Table 2: Select Behaviour in SQL Server

The examples above are quite simple and involve two sessions attempting to access the same row of data. However, select blocking can also come into play in situations that might not be expected if one were not considering how the DBMS works. Consider the following situation.

Time	User 1 Actions	User 2 Actions
1	Starts Transaction	
2		Starts Transaction
3	Updates row 2 in table A	
4		Attempts to read row 200 in table A
		table A has no indexes, so the query requires a full table scan
		U2 Is Blocked by U1
5	Commits transaction	
		Select statement completes, returning data

In this case, User 2 is not explicitly requesting the data that had been changed by User 1. However, in order for the DBMS to return that data, it must read all records in the table, including the one changed by User 1. As such, User 2 will be blocked until User 1 releases that lock.

This difference illustrates how isolation can be implemented differently. Both are valid solutions to that problem. But this is also an illustration of how different DBMS' can have very different behaviour. This behaviour will then translate into much different behaviour for your application and much more or different problems.
In general, select blocking greatly increases the frequency of locking events and, as a result, of the locking problems discussed above.

6.2 Non-Row Locking

Locking does not always involve row specific locks. SQL Server stores data in pages, with each page containing the data for multiple rows. As such, locks from one transaction may impact another transaction even though they are not accessing the same row, but because the rows they are concerned with share the same data page.

6.3 Lock Escalation

For DBMS such as SQL Server, locks are managed in memory and therefore consume memory resources. As the number of locks increase, so do the memory resources required by the DBMS to track these locks. These resources can be significant and, at some point, can overwhelm the resources available to the database. In order to manage this situation, the DBMS may *escalate* locks to a higher level. For example, if one transaction is holding many locks on a single table, the DBMS can escalate that lock to the table level. This will greatly reduce the resources required to maintain that lock and isolation. However, it can greatly impact concurrency because now the entire table has been locked.

In SQL Server, Lock Escalation occurs when:

- A statement locks more than 5000 rows in a table
- Lock resources consume > 40% of total memory

Note: Things are a bit more complicated, but this explanation is sufficient for this level of discussion

In these cases, locks may be escalated to the table level, introducing different locking events and locking behaviour than would be otherwise seen. This type of locking will also be much more intermittent and difficult to reproduce. For example, they may never be seen in testing if the data accessed never leads to large updates (>5000 rows) or the load on the system never leads to sufficiently high lock memory usage. But these situations may occur in production and lead to unanticipated behaviour and problems.

6.4 ITL Locks In Oracle

As discussed above, Oracle does not have a global lock manager, but instead stores locking information at the level of the data block. Within each data block there exists a simple data structure called the “Interested Transaction List” (ITL). The initial size of this list is determined by the INITRANS storage parameter. Space permitting, the number of slots in the list is allowed to grow up to the value of the MAXTRANS storage parameter.

Often INITRANS is set to 1. This then allows a situation to occur where row data fills up the rest of the block and, even if MAXTRANS is greater than 1, there is no room for the ITL to grow. As such, if a block contains many rows and only has 1 ITL slot, then the rows in that block can participate in only one transaction at a time. If another transaction tries to lock another row (unrelated to those locked by the first), that transaction will block. This behaviour can lead to additional locking in Oracle, which would not otherwise be expected.

7. What to do?

So now that we’ve gotten to know our enemy a bit better, how can we as QA professionals use this to avoid disaster? This section outlines the 4 complementary strategies that will help you to find these problems in the testing stages.

7.1 Use Appropriate Test Data

Using the right test data is crucial. Based on our discussion of locking issues above, it should be clear that data is important: different data will result in different behaviour. As such, it is vital that test data is chosen and designed specially with locking issues in mind.

When talking about test data, there are basically three aspects to consider: Active Data, Background, and Database Structure. All are important.

7.1.1 Background Data

Background data is data that's in the database, but is not necessarily accessed during testing. Its purpose is to provide "weight" to the database and ensure that testing validly reflects reality. For example, if your test database is 1 GB in size, containing tables with row counts in the order of 10,000 rows, but production deployments are in the TB range, you can expect to miss many things in testing. As such, the following guidelines are important to ensure that locking issues are found during testing:

- The physical size (i.e. in GB) should approximate production sizes
- Row counts in key tables should approximate production sizes
- Distribution of data within tables should approximate production data

These are important because the size and distribution of data affects the way that the database accesses data (e.g. execution plans for a given query), the amount of time that actions take to complete, and the amount of resources required to complete them. These, in turn, will greatly affect the frequency of locking events and locking problems.

For example, imagine an update statement that updates a large portion of rows in a table and requires a full table scan. If a small database was used for testing, this statement would complete quickly and would not result in lock escalation. However, with a larger database, the update statement could take much longer to execute. This could lead to a Lock Contention problem if this statement was frequent enough. It could also lead to a Long Term Blocking problem if the statement caused lock escalation.

7.1.2 Active Data

Active data is the data that will be actively accessed and utilized during testing. This data is also critically important to ensuring that locking problems are reproduced in testing. Here are several areas to consider:

- The amount of data accessed

  E.g. How many items do users really have on their home pages?
- The overlap of data accessed concurrently

  E.g. How many users are generally in each group and how many groups are there per user in the system?
- The completeness of data accessed

  E.g. How many sections are there on the homepage that could contain data?

The goal should always be to ensure this validly reflects reality whenever possible. It is generally best in a test environment to err on the "bad" side, with active data being "bigger" than 90% of real situations. But it is also prudent to not be too extreme, as this can lead to overemphasis and cause problems to appear in testing that would never occur in production.

7.1.3 Database Structure

This may seem like common sense, but it is absolutely critical that the structure of the database (including table definitions and indexes) be the same in testing as it is in production. The presence or absence of an index can completely change locking behaviour, for the better or worse.

7.2 Implement Appropriate Monitoring

Another vital strategy is to ensure that appropriate monitoring is implemented to allow identification of locking problems during testing. Without appropriate monitoring, Lock Contention and Database Deadlock problems may occur but not be noticed or identified for what they are. This is especially problematic with intermittent and timing dependent problems. Also, Long Term Blocking or System Deadlock problems may occur, but sufficient information will not be available to identify them as such and to enable diagnosis of the cause.

The exact monitoring required will depend on your application and DBMS. However, the following are crucial:

- Detect Long Term Blocking events and ensure that sufficient information is gathered. Also ensure that notification is sent out and that processes exist to ensure that these events are investigated.
- Detect Database Deadlock events and ensure that notification is sent out and that processes exist to ensure that these events are investigated.
- Capture statistical information on locking and ensure that there is a process to review this regularly and act upon problems identified.
- Capture information on application level blocking (e.g. thread contention)

The appropriate tools for such monitoring vary with DBMS and application technology.

7.3 Design Explicit Locking Tests

Designing explicit locking tests is also critical. By considering the possible locking problems and your specific application, it is possible to prepare test that will increase the chances of finding locking problems during testing. This testing, in general, will need to involve performance or load test automation in order to be successful. This testing needs to consider 3 factors.

7.3.1 Normal Activity

When designing explicit locking tests, it is best to start with tests that simulate “normal activity”. This vague term is meant to indicate the workload that the application or system is most often required to support. This would generally include short requests or actions involving a wide breadth of functionality.

7.3.2 Long Running Tasks

As opposed to Normal Activity, Long Running Tasks are the less frequent components of workload that tend to take a long time to execute. These are often batch or schedule tasks which would be run infrequently during a day or week. Examples would be data synchronization events, index rebuilds, or backups. Depending on the application or system, they may also involve end-user initiated actions that are expected to take a long time to execute. Examples would be report execution, user initiated exports, or batch imports.

7.3.3 Data Overlap

Data overlap is an important concept in defining explicit locking tests. Levels of data overlap must be explicitly designed into the tests. The reason for this is that some locking would be expected. For example, if 10 users were all attempting to update the username of the same user, blocking and contention would be expected. However, if 10 users were all attempting to update the usernames for different users, then blocking and contention would not be expected or desirable.

When defining these tests, the following sequence is recommended.

1. Start with Normal Activity
2. Add in Long Running Tasks, starting with minimal data overlap
3. Increase level of data overlap

7.4 Combine Automation w/ Manual Testing

The fourth strategy is to combine automation with manual testing. In many testing processes, Performance Testing (under which much of the above would be categorized), functional automated testing, and manual testing are completed separately. They often involve different teams and different equipment.

However, this can greatly limit the ability to reproduce locking problems during testing. This is because automated testing, even that which follows an explicit locking test plan, will almost always be narrower in scope than manual testing. Automation creates a “beaten path” where problems are quickly identified and resolved. Conversely, manual testing environments often involve broader and more variable activity. This usage is also often more valid than that which results from automation.

By combining automated load testing with manual testing, it will be possible to reproduce a much broader set of locking issues than would be possible through load testing alone. And by implementing the same monitoring used to detect locking in fully automated tests, one can be sure that problems which do occur will be identified.

Combining these in a UAT environment can also help to assess the severity of locking problems that occur. Load test automation that pushes a system to extreme levels of concurrency may result in locking issues which would not occur under normal conditions. Running load tests, at realistic levels, during UAT can help to assess whether these problems will actually occur and whether they must be fixed prior to release.

8. When To Be Concerned

Locking issues and their relative impact can be very hard to predict. However, the following is a list of some scenarios where QA professionals should be particularly concerned.

1. The application has a combination of short and long running transactions
2. An application is being ported from Oracle to another DBMS
3. An application is moving from a standalone application tier to a clustered application tier.
4. The application accesses multiple databases.

How to fix – Workarounds

This paper is mainly focused on how to identify problems during testing. Diagnosing and resolving these problems often requires code changes by developers which are beyond the scope of action for QA professionals. However, there are some configuration level changes, not involving code changes, which can help. There are also some workarounds that the QA professional should know about, since the implications can be problematic.

9.1 SQL Server: NOLOCK Hints

Most DBMS's allow for *hints* to be provided in SQL statements. In general, these do not change the end result of a statement but will alter the execution plan or some other behaviour. However, one such hint which is often used in SQL Server to work around problems of Select Blocking is the NOLOCK hint.

A NOLOCK hint tells the DBMS to relax the Isolation property and leads to "dirty reads". This allows a statement to complete without placing any locks, thereby avoiding Select Blocking and reducing the overhead on the lock manager. However, this will lead to uncommitted data being returned.

QA professionals should be aware of all places where NOLOCK hints are used within applications because they can cause strange and complicated problems. This is especially true when a NOLOCK hint is used in a query that will populate a cache. In this scenario, cache corruption can occur if "dirty data" read into the cache is then rolled back. This can also lead to variable behaviour depending on whether the data is cached or results in a database query.

9.2 SQL Server: READ_COMMITED_SNAPSHOT

SQL Server 2005 introduced the new READ_COMMITED_SNAPSHOT isolation level. This may be enabled for an entire database and will result in behaviour similar to that seen with Oracle (i.e. select statements will generally not block). This is not enabled by default and must be explicitly enabled. Since the TEMP DB is used to store older versions, this can put additional strain on the TEMP DB. However, this option is overhead is generally manageable and well worth the benefit of reducing locking issues.

9.3 SQL Server: Disable Lock Escalation

Lock escalation can be disabled using the TraceFlag-1211 or TraceFlag-1224 trace flags. However, these should be used with caution as they can lead to other problems related to exhaustion of memory.

9.4 SQL Server: Avoid Full Table Scans with Indexes

If common queries are leading to full table scans on SQL Server without READ_COMMITED_SNAPSHOT, this can increase the chances of locking problems. This is because the full table scan will access every row and will be impacted if any row is locked. Creating an index that is included in the execution plan will reduce the incidence of locking.

9.5 Oracle: Size INITRANS

Problems relating to ITL locks can often be avoided by setting a larger INITRANS parameter. This database level change can often be implemented during installation of your system and can be implemented after the fact (though this requires a rebuilding of the related objects). The proper size depends on many factors and a larger INITRANS will result in less compact data storage. As such, these changes should only be done in close consultation with a trained DBA.

Code Generation for Dummies

Matthew Fowler, matthew.fowler @ nte.co.uk
New Technology / *enterprise*, http://www.nte.co.uk

Introduction

When I was a lad, code generation referred to the final phase of a compiler. Having digested your source code, the compiler would pump out the "code" for the target machine. At that time, "machine" meant a CPU instruction set - so this was machine operation code being generated.

Having learnt all compiling languages like FORTRAN and PL/I, my second job - way before the PC, let alone the dreaded CASE tools - was to build a code generator to create CRUD applications. Back then, it was practically unheard of to do this. These days, 'code generation' is so well established that Wikipedia has a new meaning for it - i.e. "source code generation" rather than "machine code generation"... and it has all sorts of new terms and ideas swirling around it.

It turns out that there are fundamental similarities between the old and new versions "code generation, and comparing them helps us understand the similarities. It is a good place to start to understand today's code generation landscape.

I'm going to start by drilling down into old "code generation" and compare it relationship to DSLs and modelling ... and establish the similarities between all three. Using this common basis, I'll explain the reason behind the recent fashion for "XML with everything" and the de-facto standard approach to defining languages in XML. Finally, I will actually describe the first steps in code generation.

Something Old

A compiler runs through a number of phases in processing an input language program:

- *lexical analysis*, which turned a character stream into a 'token' stream.

  For example, if we write "i = 0;" in C or its successors, this turns into the tokens

      [IDENTIFIER(i)] [EQUALS] [INTEGER(0)] [SEMICOLON].

- the preprocessing phase. Some languages, notably C, have preprocessing that works on tokens, so they had to go next. Preprocessors could include source files, paste tokens together or turn arguments into a complete new sequence of tokens.
- *parsing*, which analyzed the tokens according to the grammar of the language and produces an Abstract Syntax Tree (AST): the "abstract syntax" is the grammar free of syntactic confetti like ';' or '.', and it's a tree because there is one root - the program - with the contents of the program beneath it as AST nodes. In an OO world, the AST nodes will all be derived from a Node base object but have specific types for the grammatical constructs - IfStatementNode, IdentifierNode, IntegerConstantNode.
- semantic validation and optimization. This worked directly on the AST, checking out that it made sense (e.g. referenced variables were declared if necessary) and optimizing it.
- finally - at last - the actual ***code generation*** bit, which wrote out the machine code or some equivalent intermediate form.

The two main tool-sets I know of to help with compiler construction are the LEX/YACC pair from UNIX and ANTLR. Both of these help with the lexical and parsing phase, and use a specification of the syntax/grammar to do their work.

In computing terms, many of the ideas about compilers are not old - they're positively ancient. Descriptions of syntax still derive from BNF (Backus Normal/Naur Form), dating back to the 1950's; parsing techniques were regularized in the 1960s.

Summary for dummies:

> Language processing, including code generation to machine code, is well-established and old hat. The main structural phases are lexical analysis, parsing and code generation.

Something New?

Now, if we're fashionably up-to-date, we'll shun general-purpose programming languages ('GPLs') like Java or C# and do development by creating a model or writing a program in a Domain Specific Language ('DSL'). So is this new? Let's look at what happens in language terms when we use a DSL, taking regular expressions as an example.

The "regular expressions" DSL was one of the languages built in the 1970's to assist UNIX developers. The initial raison d'être for UNIX was typesetting at Bell Labs (it took a slight detour along the way) so mini-languages were created around the notion of text manipulation - sed, awk, grep and regular expression's are the ones that roll off the tongue.

A regular expression is a series of commands to find a substring of a larger string based on its shape rather than the exact text - so it was pattern matching for text. A regular expression is made up of a number of actions - basically telling the interpreter what to do to find the string. For example:

^	Match the beginning of the line - i.e. what follows must be at the start of the line
21:	Match the string "21:" exactly.
[a-z]	A range, given in the [] brackets, in this case it is from a-z.
[^W]	Match any character except upper-case W.
+	Using the preceding regular expression, match one or more of them.
.	Match any character.
*	Using the preceding regular expression, match zero or more of them.

Most modern editors allow you to search files for a regular expression and they're very useful in certain situations. For example, we've done a lot of looking at logs recently. In thousands of lines, we often want to find where the initialization is finished for a particular run, so there is a line like:

```
21:19:49 132.940.838 Brokerage getProxy Working at 21:19:49
```

Searching for an exact string won't work here because the timestamps change. To find where the system started working around 9pm, we can do a search in the editor using a regular expression like

```
^21:[^W]+Work.*
```

The '^' matches the beginning of the line and then the characters "21:" select the 9pm timeframe. The [^W]+ skips one or more characters that are not 'W', followed by "Work". '.*' matches the rest of the line.

Let's come back to the idea of a regular expression as a series of commands. This is a fundamental pattern in software: *we're telling a processor what to do*. You could also write the processor in Java, in which case you would control it using an API. Something like this:

```
MatchSequence matcher = newMatchSequence()
                           .startLine()
                           .literal("21:")
                           .oneOrMore( new AnythingBut("W") )
                           .literal( "Work" )
                           .restOfLine();
```

(If you know regular expression, please forgive the conversion of '.*' into 'restOfLine' - it would be strictly correct as a 'zeroOrMore' plus 'anyCharacter' combination.)

While lengthy regular expressions demand patience and humility to debug, in their domain (pattern-based text searching) they are sharper and faster than the general-purpose alternative. This is characteristic of a good DSL: its conciseness aids thinking because there is a direct correspondence between how an expert thinks and the symbols of the DSL. This means that the **form** (the syntax and its resonance with the domain expert's concepts) and context (for example, the spreadsheet presentation of Visicalc) are vitally important to the success of a DSL. The rules for writing correct programs are often called the ***concrete syntax***; and it has its own rules of composition just as the ***abstract syntax*** does.

As you have probably guessed from the section title, the processing of DSL's proceeds exactly as for general purpose languages. The lexical analysis splits the text based on the special characters:

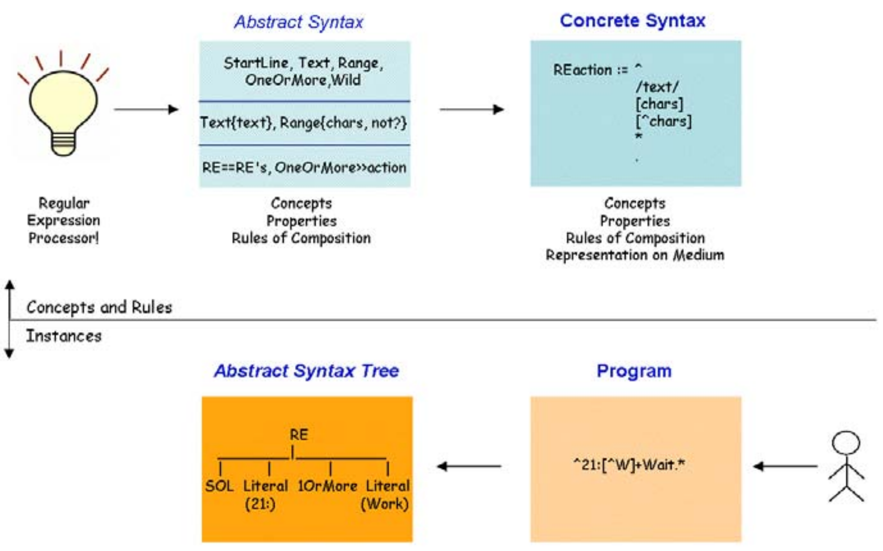
```
STARTLINE TEXT LEFTBRK UPARROW CHARS RIGHTBRK PLUS TEXT DOT START
^         21:  [       ^        W    ]        +    Work  .  *
```

The parser creates an AST that is similar to the Java code given above:

```
                      RegularExpression
                               |
    ______________________________________________________
   |            |              |            |      |       |
StartLine Literal("21:")  OneOrMore Literal("Work") RestOfLine
                               |
                       AnythingBut("W")
```

I find it helpful to think of the AST as the in-memory representation of the instructions for the next processing step. We're interested in code generators, but the regular expression handler in my editor is interpreted; the same AST works equally well. Each node in the AST is an instance of a class, with qualifying attributes (e.g. "21:") and possibly children - for example, the OneOrMore class must have something to work on, which is represented by a child instance.

Generalizing from this example, we use the following terms about abstract syntax:


Abstract Syntax
StartLine, Text, Range, OneOrMore,Wild
Text{text}, Range{chars, not?}
RE==RE's, OneOrMore>>action
Concrete Syntax
REaction := ^ /text/ [chars] [^chars] * .
Regular Expression Processor!
Concepts Properties Rules of Composition
Concepts Properties Rules of Composition Representation on Medium
Concepts and Rules
Instances
Abstract Syntax Tree
RE
SOL Literal (21:) 1OrMore Literal (Work)
Program
^21:[^W]+Wait.*

The top half of the diagram is about the design of the language. The language creator has a bright idea about a processor for a domain, which he or she elaborates as follows:

- The domain is based on some **concepts** (e.g. start of line) and either a description of information or operations to do on those concepts. For the regular expression language, the "operation" is implied in all the AST nodes - they all request a match of the string.
- Concepts can have some **attributes** to supply additional information. For RE's, these are the string to match ("21:"), or the available character range.
- A program is a **composition** of concepts and attributes. For the processor to know what to do, the "OneOrMore" node will need a child expression. This sort of composition is apparent from the AST, but others are more subtle, such as not defining a variable with the same name twice.

This is the abstract syntax. The concrete syntax also has concepts, attributes and composition rules, plus the additional syntactic framework so it can be represented in a particular medium, such as the braces in the 'class {...}' construct for the written form of a program.

n the bottom half of the diagram, the programmer writes the program in concrete system which is transformed into an AST.

Summary for dummies:

- an abstract syntax comprises **concepts**, **attributes** and **rules of composition**
- a concrete syntax has the same elements as the abstract syntax, with additional markings to map the abstract syntax onto the program medium like a text file
- there's no difference in <u>processing</u> DSLs (Domain-Specific Languages) and GPLs (General-Purpose Languages); the difference is in the form and representation of a specific domain's concepts and actions
- the acceptability of a DSL in a particular domain is determined by its form and presentation (because the underlying concepts and compositions are likely to be similar between different DSLs for the same domain).

Advertisement - Software Testing Analysis & Review Conference - Click on ad to reach advertiser web site

SOFTWARE TESTING
ANALYSIS & REVIEW

The Greatest Software Testing Conference on Earth

- 34 In-depth pre-conference tutorials on how to be more efficient and effective
- 40 Concurrent sessions on how to turn challenges into opportunities
- 5 Keynote presentations from top testing experts with experience in down times
- Networking opportunities to see how others are handling economic challenges
- Largest Testing EXPO event to help you find solutions
- Testing & Quality Leadership Summit on Friday to help you become a leader in the quality movement

TESTING & QUALITY LEADERSHIP SUMMIT

A Co-Located Event New for 2009!
Testing & Quality Leadership Summit
Friday, May 8, 2009

SOFTWARE QUALITY ENGINEERING
CONFERENCES
www.sqe.com

www.sqe.com/go?SE09M&Tad REGISTER EARLY AND SAVE UP TO $300!

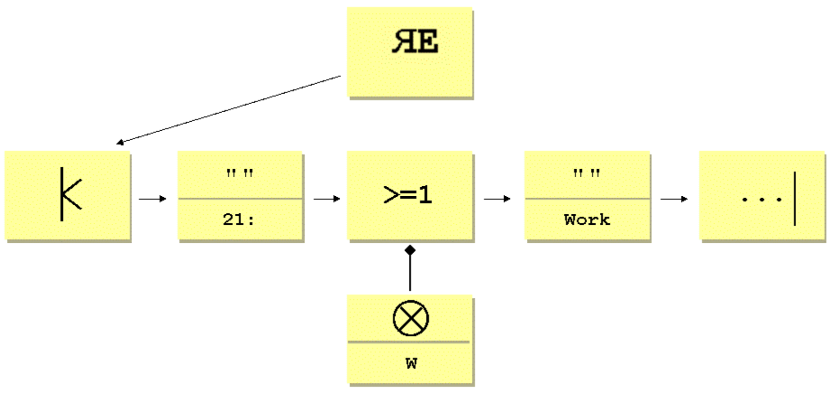
Models and Graphical DSLs

We introduced 'models' at the end of the last section, then promptly ignored them. Now it is time to return to them, and address the terminology minefield, which has generated much confusion and heat in the last few years.
To a lot of people, models mean "UML models". The problem with this for our discussion of code generation is that UML models have historically been more of a means of communication than a rigorous specification of what a program has to do. As our interest is in code generation, I will use the term "model" to mean something written in a ***Graphical DSL*** that is precise enough to generate code from; we could also be more precisely about the regular expression grammar and call it a ***Textual DSL***.

There is a growing trend in the Microsoft and Eclipse camps to use "DSL" to mean "Graphical DSL", but this is strictly incorrect.

A graphical DSL has an editor with a range of objects that can be drawn on the modelling surface. Typically you can add names and other details on the drawn object; for a 'Class' we can add fields and methods for example. There are also relationships between the objects, often expressed as lines but sometimes by other graphic devices, like attachment or nesting one object within another.

Graphical DSLs provide picture symbols to represent the use of a concept rather than a text representation, presumably because 8-year-olds and business sponsors find these easier to understand. So, rather than writing "^21:[^W]+Work.*" or the AST equivalent, we might construct the graphical DSL as follows:

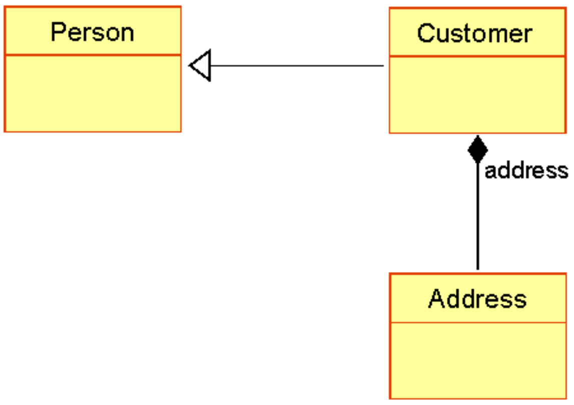


The graphical DSL can throw away the syntactical clutter, like '[]' for the character range, because it is implied by the symbol. However, the composition still has to be expressed somehow: in this rendering, it is explicit using arrow to imply order and the container symbol for the parent-child relationship. This is an example of concrete syntax mapped onto a graphical medium. The above picture now looks very like the AST we drew out textually ... which was

```
                       RegularExpression
                               |
            _____________________________________________
            |          |            |            |          |
        StartLine   Literal     OneOrMore     Literal   RestOfLine
                    ("21:")         |         ("Work")
                                    |
                             AnythingBut("W")
```

This is no accident: graphical DSLs allow users to use concepts, specify their properties and compose them into composite structures, just like DSLs do.

To briefly come back to UML models, and address their relation to graphical DSLs. The basic diagrams in UML modelling are DSLs in their own right, and in particular, the class diagram defines the information behind a C++ or Java program pretty well. At this basic level, "code generation" means converting UML to C/Java/whatever syntax; round-tripping means going back from code into a UML diagram. For example, this diagram

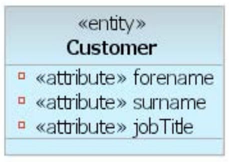


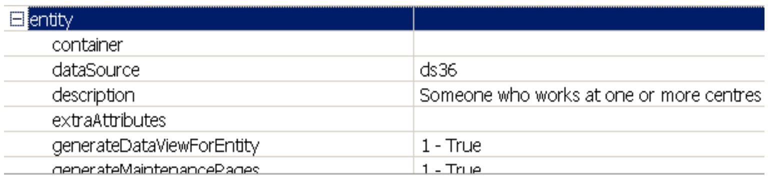
is equivalent to the Java code

```
class Person {
      ...
}
class Customer extends Person {
      Address address;         ...
}
class Address {
      ...
}
```

The class diagram of UML gives us another naming issue. Strictly speaking, it is a graphical DSL, but its domain is so generic - the description of classes of objects and their relations - that for most developers these days it is more like the data definition part of a GPL.

UML can be made more interesting by adding "profiles", which specializes all the modelling elements with a user-chosen "stereotype" name (e.g. in entity-relations modelling, we would use `<<entity>>` and `<<relation>>` stereotypes) and related tagged values like "description". For example, here is a stereotyped UML class and some specific tagged values for the stereotype:





entity	
container	
dataSource	ds36
description	Someone who works at one or more centres
extraAttributes	
generateDataViewForEntity	1 - True
generateMaintenancePages	1 - True

UML with stereotypes is the quickest approach to building a graphical DSL if your company has a UML tool, but it has drawbacks: the rules of composition that you would expect in a specialised graphical DSL cannot be specified - they will default to the UML rules; also, all the native UML information will be present in the modeler which is distracting if it is not relevant to the domain.

Graphical editors will have a serialization format, which is now a textual representation of the "program". For UML, this is XMI (XML Metadata Interchange), which is as baroque as its name implies. In Eclipse, models can similarly be serialized in XML dialects.

Summary for dummies:

- A model, or graphical DSL, uses a visual representation of its concrete syntax rather than a textual one.
- A model often suggests the structure of an AST, and for processing purposes a model is converted to an AST. So for code-gen, Model ==> AST.

XML and DSLs

The AST is the Great Divide of Code Generation: on our left is the expression of the intent in the source language - GPL or DSL, textual or graphical - and its transformation into an AST; on our right is the production of the code or whatever else we want to do with the AST. The AST is the central point of leverage.

Before we press on and talk about code generation, there are some important points to note about the AST. The starting point is that an AST is isomorphic to an XML document. Here comes the now-famous regular expression example, recast as an XML document using attributes for the embedded details:

```
<RegularExpression>
       <StartLine/>
       <Literal text="21:"/>
       <OneOrMore>
       <AnythingBut characterRange="W"/>
       </OneOrMore>
       <Literal text="Work"/>
       <RestOfLine/>
</RegularExpression>
```

This XML could be a serialization of the AST ... or it could be written directly, with the XML dialect as the domain-specific language. Here's the entity-relational model we discussed earlier, recast as an XML 'program':

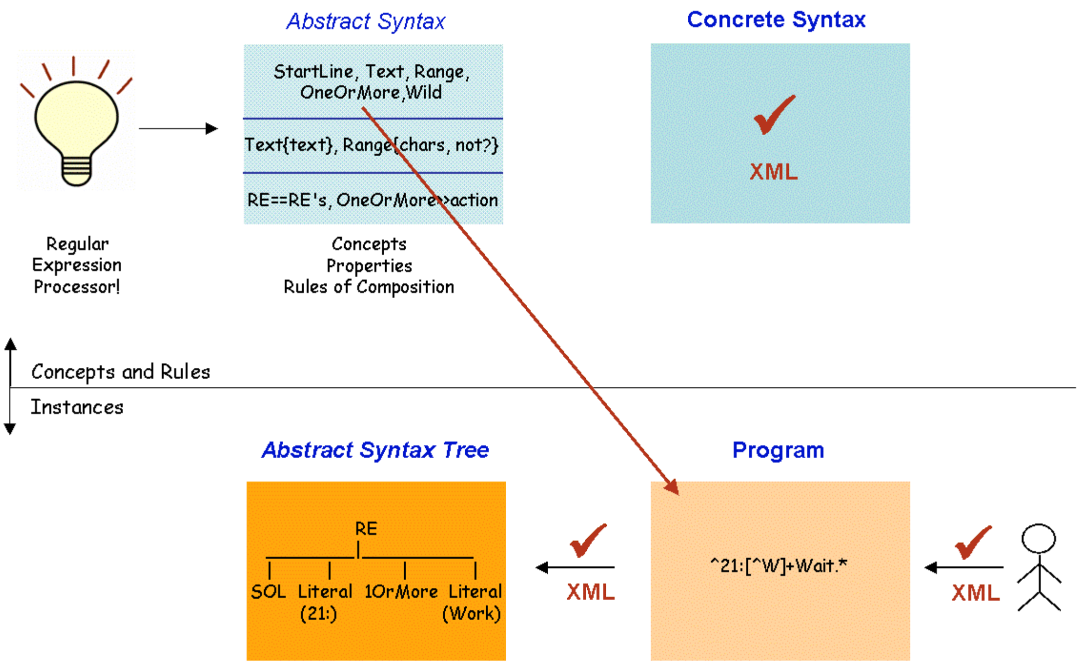
```
<entity          name="Customer" dataSource="ds36"
                 description="Someone who works at one or more centres">
    <attribute   name="forename" type="String" />
</entity>
```

Developers have been using XML dialects as ad-hoc DSLs for some time. (While the "Domain" in "DSL" often has a connotation of "business domain", there is no reason this can't apply to technical domains, which is where this sort of DSL approach is used most.) In other words:

- the concrete syntax is mapped to XML syntax;
- the concepts are used as XML element tags;
- additional information is mapped to attributes;
- and the composition capability restricted to nesting which allows you to build up the AST tree.

This gives a natural or "bare" format we'll call ***AST-XML*** (although I'm sure someone's dreamed up a better name, if only I could find it): there is no additional syntax or meta-information in the XML model, making it is all a direct representation of the AST in XML.

So here is our "language creation process", with the XML short-cuts:


Abstract Syntax
Concrete Syntax
StartLine, Text, Range, OneOrMore,Wild
Text{text}, Range{chars, not?}
RE==RE's, OneOrMore->action
XML
Regular Expression Processor!
Concepts Properties Rules of Composition
Concepts and Rules
Instances
Abstract Syntax Tree
Program
RE
SOL Literal (21:) 1OrMore Literal (Work)
XML
^21:[^W]+Wait.*
XML

- the additional syntax in the concrete syntax is taken care of by XML
- the training of user's regarding the syntax is taken care of by XML if the user is a programmer
- the syntactical analysis for producing the AST is handled by standard XML tools
- the developer still has to define classes for converting the XML stream into objects.

XML-based configuration in the last few years has pretty much standardized on this approach - e.g. Spring beans definition, for defining beans to add into your program.

```
<beans ...>
  <bean id="customerSimulator_ActionsBean"
     class="com.equity.customer.CustomerSimulator_Actions"
     init-method="Gsb_InitiatePulsesAndListener"
     />
</beans>
```

The intent of this program is to inject an instance of the CustomerSimulator_Actions class into the program. Spring Beans therefore is a DSL for defining class instances and their interrelation.

To understand how simple the AST-XML approach is, contrast it with other approaches that include this data, such as this from XMI version 1.1

```
<UML:ModelElement.taggedValue>
       <UML:TaggedValue tag="BusinessName" value="a3"/>
</UML:ModelElement.taggedValue>
```

Most of this is describing XMI itself. The useful information would be written `BusinessName="a3"` in AST-XML. The lack of superfluous stuff makes it easier to read and to transform from or generate code.

We cannot leave this subject without mentioning Ant's clever approach **to *implicit AST instantiation***. ***Ant*** allows developers to write their own plugins as Java classes which are invoked by XML in the build file. The normal approach to converting XML elements to model objects is to use SAX to read the XML and then instantiate objects to represent the XML elements.

Instead, Ant uses implicit instantiation using introspection and naming conventions, which works as follows. There is an AST node for regular-expression matching condition - i.e. there is a `<matches>` element in the XML. This can have a nested element `<regexp>`, which defines the RE to match. If the class for the `<matches>` element is Matches, then for this to work it must have an addRegExp() method - i.e. 'add' plus the capitalised name of the nested element. There must also be a parameter (often the same name as the nested element - e.g. `RegExp)`. So when Ant sees the `<regexp>` element, it finds the addRegExp() method, creates a an instance of the class, then calls addRegExp() passing in that method:

```
matchesElement.addRegExp(new RegExp)
```

The same convention-based approach is used to set the attributes. This time, mapping of attributes to the AST node is by setX() methods, so the XML pattern="^21[^W]+Work.*" results in the call

```
regExpElement.setPattern("^21[^W]+Work.*")
```

Implicit AST instantiation separates the concerns - of the construction of the AST, from the definition of the class itself - so generating the AST-XML is equivalent to generating the AST. The combination of AST-XML and implicit AST instantiation removes all the syntax and grammar work that used to be required to define a language.

Summary for dummies:

- The AST is the Great Divide of Code Generation, joining the expression of intent - the language - to the processor that is going to carry out the intention.
- An AST is isomorphic to an XML document; there is a natural "AST-XML" mapping.
- Technical language developers are starting to use XML dialects as DSLs.
- AST-XML plus implicit instantiation removes all need for the language designer to implement syntax and grammar processing

AST Representation

Code generators (or interpreters) are programs and so there needs to be a generate-time representation of the AST. This is usually implemented by defining classes (in the same as the generators language - Java, C# etc.) for the concepts, fields for the properties and lists for nested children:

```
Class Entity extends ASTNode {   // concepts become classes
 String        name;    // additional attributes become fields
 String        dataSource;
 ArrayList<Attribute> attributes; // children become lists
 ArrayList<ASTNode>  children;  // alternate view of children in order
}
```

```
Class Attribute extends ASTNode {
 String       forename;
 String       lastName;
}
```

For small applications, it is most convenient to define these directly in Java. When you have more than 50 classes or so, it is easier to generate the classes for the generate-time representation. This not only saves coding time (you can generate about 50% of the code at this level); it is also easier to change when the structure of the generate-time classes. The definition of the classes in the AST is what the UML folk call a ***MetaModel*** and should precisely define the classes for the concepts in the DSL or model. Here are the classes above turned into a BusinessObjects meta-model, which includes the Entity amongst others:

```
<MetaModel     name="BusinessObjects">
  <ASTClass    name="Entity" >
    <Property  name="name" /> <!-- String is the default -->
    <Property  name="dataSource" />
    <List    name="Attributes" type="Attribute" />
    <!-- list of children automatically generated on all ASTClasses-->
  </ASTClass>
  <ASTClass    name="Property" >
    <Property  name="forename" />
    <Property  name="lastName" />
  </ASTClass>
<MetaModel     name="BusinessObjects">
```

In our experience, building a meta-model is made much more valuable if the user (i.e. programmer, or end-user, as appropriate) documentation is written along with the definition of the meta-model. The concepts in a domain are absolutely key to most projects' well-being, but their meaning has a habit of shifting over time; the documentation anchors the meaning to the original definition and also serves to communicate between project members.

The final point to note about AST classes/meta-models is that the meta-model (apart from that elusive meaning!) are much more stable than most other artifacts. Technologies come and go, more features get added into the code generation stage, other architectures are addressed ... but the meta-model changes very little over time.

Summary for dummies:

- A meta-model is a definition of the concepts, attributes and composition of the AST classes. It can also include validation and 3GL code for operating on the nodes.
- Meta-models define the abstract syntax rather than the concrete syntax. Given our focus on code generation, the meta-model is more useful than a grammar definition like BNF.

System Generation

You may be slightly nervous by now that, in a paper about "Code Generation", we will never get to a section about it. And you would be right!

We believe it is more correct to talk about "system generation", because these days development is just as much about specifying annotations and configuration as it is about writing code. This is no problem for modern generation systems that use the templating approach, because these can which can generate any type of output, not just code. There are a number of levels to system generation, which appear in the following sequence.

Level 1 - Single File

Generation of a single file of code is relatively simple, using the templating idea, where we insert variables into some constant text representing the type of file generated. This level of code generation is also called ***Model-to-Text*** or ***M2T***. In theory, you could of course write a generator in Java or C#, and this works for small applications. In practice, this approach is ugly and inconvenient, which is why specialised languages have emerged for this job.

There are more generation engines than you can count. For the Java world, Velocity is very convenient - it seems to be the choice for model-driven engines.

Different generation 'engines' have various ways of specifying the variable bits in a template text. So let's look at a Velocity template as an example:

```
 1                class $name
 2                {
 3 #foreach( $property in $propertyList )
 4    #if( ! $property.type )
 5      #set( $property.type = "String" )
 6   #end
 7                  $property.type $property.name;
 8 #end
 9 #foreach( $lst in $listList )
10    #if( ! $lst.type )
11      #set( $lst.type = $this.capitalise( $lst.name ) )
12   #end
13                  ArrayList<${lst.type}> $lstName;
14 #end
15                }
```

There are two streams here: on the right is the template for the textual output; on the left are Velocity directives to guide the processing. Quick notes:

Line 1	`$name` is a reference to a field in the "context object" - which is the AST object this is template is being run for. The name picked output by introspection, so $name will take on the values *name* and *dataSource*.
Line 3	`#foreach( $x in $someList )` - nested XML elements like <property> and <list> are made available through the '...List' object - i.e. propertyList and listList in this case.By convention, nested <property> items are collected into the `propertyList` object.
Line 4	`#if( ! $property.type )` - Velocity's if statement. The '!' means *logical not*, and the following expression uses existence as logical true.
Line 5	So if the property type is not defined, then this statement sets the property type to the default, which is "String".
Line 6	This is an output line and results in e.g. "`String name;`".
Line 13	`$this.capitalise()` - Velocity uses Java introspection to pick out available methods on the context object. "`capitalise`" is a standard library method.

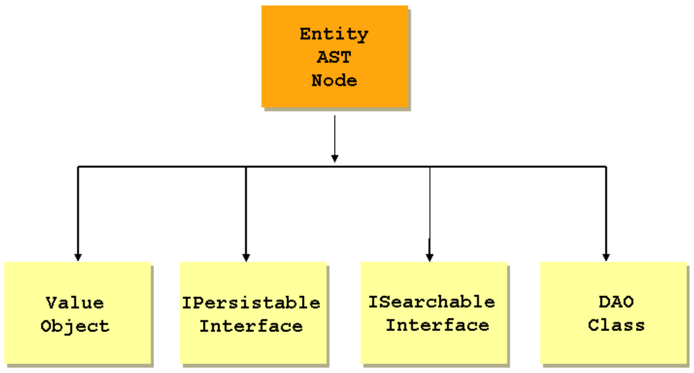
The sort of generation shown here is exactly what some people mean by the term "code generation" - converting a model to a programming language like C++ or Java. The following sections describe the elaborations on single-file code generation.

Summary for dummies:

- To generate simple text files, generators use a templating approach. The ***template*** has a mix of constant and variable text; the variable portions are retrieved by evaluating an expression using values from the ***context object***.

Level 2 - One Modelled Element, Multiple Files

One of the slogans about code generation is that it can raise your level of abstraction. Here is our example of this idea. It occurs when one modelled element (AST node) creates more than one output file; so the modelled concept (e.g. entity) is more abstract than the realization in Java or C#. Here is the generator creating a range of files required to implement the Entity concept:



The generation system must provide ways to specify this. There must be a *workflow coordinator* that runs the generator multiple times on the same context object using different templates. In a Java environment, this can be done by invoking an Ant or Maven build for the component: it can run the code generator multiple times on the same object using different templates.

Ant build for Entity component → multiple invocations of file generator

Note that, if we use Ant or some other general-purpose build coordinator, then we can not only generate files, we can also compile classes, create Jars, run tests and so-on. This means that the generator is not just generating code; it's generating coordinated executables like Jars or Wars in the Java environment.

Level 3 - The Complete Tier

If we have a model for one layer of architecture, like the persistence layer, then the next step is to run a build for every object in the AST. This will generate a complete architectural tier.

The mechanics are to use a ***tree-walker*** to visit each node in the AST, bottom-up, to build smaller components first, and then assemble them into larger components. At each node, the tree-walker runs the build on the component if possible. For example, methods quite often generate 'snippets' as part of their build; class-level objects produce class files; at the next level up, the Jar can compile all the classes and collect them into a Jar; and so on to application and deployment.

In other words, using the tree-walker, it is possible to do a "manufacturing assembly" of the complete model to produce a complete tier of the target architecture.

If the tree-walking is part of the generator framework, the individual component can just "build itself", and leave it up to the framework to do the assembly.

Interlude - Reverse-Engineering and Business Logic

With time, we gradually increase the amount of a system that we can build. In 2001, getting 50% code generation was pretty good; now, 95-97% generation is possible. However, there is still hand-written code to weave into the generated framework.
First of all, let's talk about the features, then we'll discuss implementation. The most common approach in supporting business logic is to mark, in the template, an area that is to be preserved when the file is regenerated:

```
int myMethod( String s ) {
   /********** special section begin ***********/
   return 0;
   /********** special section end ***********/
}
```

When the method is first generated, the boilerplate code "return 0" is inserted; thereafter, any changes are preserved when the method is regenerated. This looks like a fairly easy facility to implement at first sight. But what happens when the method name or signature is changed, or the method is moved to a different class? Most systems don't handle this too well, and it causes enough problems to give code generation a bad name.

The general solution hinges on marking every model element with a unique ID in some way - which is fine because all modelling tools seem to do this now, although it does give problems if other approaches are used. Then the markers can incorporate the UID:

```
int myMethod( String s ) {
   /********** UID: 1234-5678-90AB begin ***********/
   return 0;
   /********** UID: 1234-5678-90AB end ***********/
}
```

To handle moving methods between files, all the special sections for all the generated code must be extracted before generation begins and then re-inserted as part of the generation - possibly into another file if the method has moved. Also, to handle renaming of classes, the generator must keep track of files as it generates them and remove any files that are no longer required by the build. Files normally become obsolete because the model changes, but it can also be because a change in naming standards has been implemented.

One of the advantages of code generation is that you can very easily generate complete trace information - all methods entries and exits, plus logging parameters and return values - for a method (we tend to do this during debugging, then remove it on going to system test to get the best performance). This is invaluable in distributed programming for example.

As an interesting side-note, our experience is that it is much easier to implement this type of work using text searching than parsing the language files. Given the right text-handing tools, it only takes a handful of statements to parse generated files and pick out the special code.

A final point, to do with encouraging programmers to keep their hands off the generated code. First, you can collect all the infrastructure into a Base class and have the user-viewed class inherit off that; this reduces the temptation for the programmer to start changing generated properties. This seems to be a well-accepted approach. Another clue is to separate the generated-only source and put it into a different directory tree from the implementation classes.

Level 4 - Language Techniques

Writing templates for individual files on a given generation target can take you so far - up to around 50 or 100 templates, the refactoring and maintenance load is tractable. To build more complicated architectures, different techniques are needed; we now address the specific language features for handling code generation.

We have already seen a very important - if mundane - technique for writing templates, namely ***indentation***. This becomes vitally important as the generation logic and the output itself get more complex. Generating JSPs for web pages is the worst: it is easily possible to have 5 levels of nesting in the Velocity script and 15 in the output. Without indentation, this becomes impossible.

One of the big issues in generation is the support from the underlying language of the AST. For example, if the AST classes are implemented in Java, can you get ***support from Java***? Velocity provides features to evaluate expressions and set properties on the underlying Java objects. As Velocity is open source, it is easy to extend it to create additional objects - i.e. objects that weren't created as part of the AST - to hold snippets of text, lists (e.g. $newArrayList to create an ArrayList object) and so on. This general approach - to give unrestricted access to the features of the AST language - avoids the need to recreate Java functionality in Velocity and makes the approach more acceptable to developers used to Java.

.. and so what is the split in responsibility between the M2T language and Java? The simple answer is that, once the generation system grows past a few hundred templates, you'll take all the help you can get!

The OMG has a language in this category called ***MOF2Text***, the 'MOF' part of that being UML model elements (". This is not widely accepted to our knowledge and this must surely be in part to its basis on "MOF" objects, which - like the more exotic products of atomic collisions - are not often seen in the wild. The MOF2Text 'library' contains around 20 methods; the Java library contains thousands of classes and over 10,000 methods. The ability to leverage a well-known and popular language compared to using a small and immature language for the underlying objects makes it clear why the Velocity approach is more powerful.

There are further issues that arise in generating large systems which we do not have space to discuss here. In summary, these tend to focus the language and its utility methods on the domain of generation, which is basically text handling and AST walking.

Level 5 - Model-to-Model (M2M)

The main technique to handle large-scale architecture, and also abstractions, is a ***Model-to-Model*** (or ***M2M***) approach. The original motivation for this, as expressed in the OMG's MDA approach, was to create a series of "platform-independent models" (PIMs), each of levels decreasing abstraction, until finally the "platform-specific model" (PSM) was reached, and which point Model-to-Text could be used.

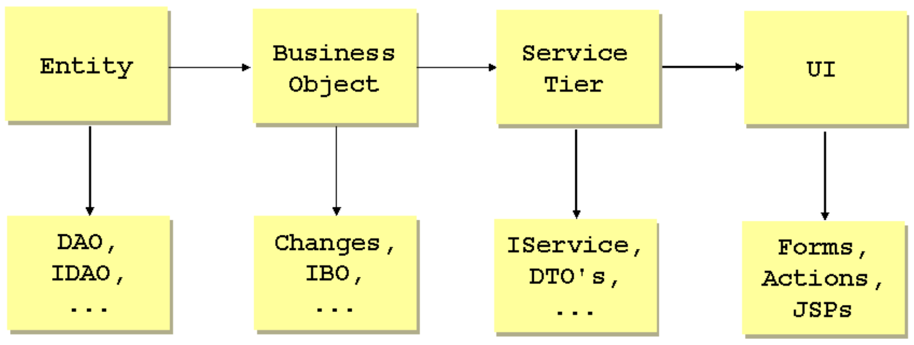
Our experience, from building generation systems comprising thousands of templates, is that this theory is not borne out in practice:

- the "series of PIMs" does not occur in practice
- we have never generated a purely abstract intermediate object (in an intermediate PIM)
- model-to-model transforms only make sense when they enhance the AST rather than creating new models - this is better done by writing XML and reading that in.
- the "platform-specific model" is better handled by a workflow component that generates multiple output files for one object - see the "Level 2" section above - so there is really no need for a separate PSM.

However, we do find practical applications of the M2M approach to enhance the AST (i.e. the input model) in these situations:

- generate smart defaults. For example, in a persistence application, generate primary keys and onLoad/onStore methods if they have not been specified in the model;
- generate Java using a (platform-specific) model of Java (i.e. generate "<class.../>" output rather than "class { }"). This allows different flavours of class{} to be generated and reduces the effort to maintain the mapping to Java.
- to create new tiers and other ***architectural patterns***, described next.

When we build architectural tiers, we need to project one tier into another. For example, in creating CRUD pages, we want to "project" the data tier into pages - typically one user-facing page per entity. The same occurs with normal Java patterns such as the facade pattern or master-worker patterns. This overall approach is recursive: it can be repeated throughout the tiers. For example, to build an end-to-end CRUD pattern, we do this:



The entity generates its own implementation objects (DAO etc.) as described previously. It also generates a related business object, that is a simple projection of the persistence tier object to the business object layer. This object sits in the mid-tier and has the main business logic and data transformations from the service interface requirements back to the database.

The pattern is repeated in the business object: it creates objects for its own implementation, then projects itself into the service tier by generating an object for doing CRUD operations on the business object - with create, read, update and delete operations on it! And so on.

There are a number of languages around to do this - QVT and ATL for example. Given our view of special purpose languages, we feel a much better approach is to use XML. Here is an example of a model-to-model generation, simply writing the class-generation example above as a model-to-model transform to create the data transfer object.

```
                <class                                   name="$name" >
#foreach( $property in $propertyList )
    #if( ! $property.type )
        #set( $property.type = "String" )
    #end
                  <field                                 name="$property.name"
                                                         type="$property.type"
                                                         />
#end
                </class>
```

This approach is very powerful because

- it separates the AST-XML generation from the reading in to the AST (which will be the same as the original AST read-in) in an XML and subsequent model-to-text generation. All these concerns are separated into independently-maintained parts
- it is recursive - a model element can project itself through multiple tiers, just as is required by the CRUD pattern. In an enterprise architecture, cross-tier patterns like this expand the number of model elements in the build by a factor of 20 (for a lightweight stack like Hibernate, Spring, JSF and Apache Trinidad) and 50 times (for J2EE).

Level 6 - System Generator Product Lines

There are additional code generation techniques that become important when building a range of product generators - like aspects, using symbolic names via the AST, overridable objects for generating the small details in a number of technical areas, swapping technologies around, and allowing for customer enhancements that easily change the core of the generation.

These are not general techniques in use today, so we will leave them for another time - you will be happy to hear.

Conclusion

We have talked a great deal about the techniques of Domain- and General-Purpose Languages, Modelling and Code Generation. Using these techniques as a platform, how does our view of software development change?

When my company first started doing system generation, the largest cost was the simple mechanics of writing lots of transforms. With better techniques, this cost has reduced significantly, so the largest cost now is understanding the platforms we are targetting. In other words, we find that most of what we do is get the ***know-how*** for the platform, understand the patterns of *best practice* and provide sample ***architectures***. Writing the transforms for all of this is relatively straightforward.

However, we also note that most of the ***concepts remained unchanged*** - some of the definition files haven't changed in 5 years. This means that models we created back then will still work today.

Development groups of all sorts have exactly the same issues of learning new areas and migrating. So whereas we used to see ***automation*** and ***quality*** as the biggest benefits of code generation, now the main benefits are in reducing the amount of learning and the risk of creating a bad architecture and - when the time comes - the cost of migrating.

The significance of the interest in DSLs is that there will not be a universal successor to 3GLs of today. So the best hope for advancement in delivering business value effectively is the use of modelling and DSLs - be they technical or user-oriented - integrated with 3GL code, and projected forward into model-driven testing and deployment. For implementing models and DSLs, and integrating them with 3GLs, there will be a growing role for code generation.

References

ANTLR http://www.antlr.org/

ATL http://www.eclipse.org/m2m/atl/doc/

awk http://www.gnu.org/software/gawk/manual/gawk.html

BNF *Backus Naur Form* versus *Backus Normal Form*

http://compilers.iecc.com/comparch/article/93-07-017

Code Generation http://en.wikipedia.org/wiki/Source_code_generation

grep http://www.opengroup.org/onlinepubs/007908799/xcu/grep.html

Lex http://dinosaur.compilertools.net/lex/

MDA http://www.omg.org/docs/omg/03-06-01.pdf

Programming Languages - Peter Wegner "Programming Languages - The First 25 Years"
http://www.cs.ucdavis.edu/~su/teaching/ecs240-08/readings/PLHistoryDesign25yrs.PDF

QVT http://www.omg.org/docs/formal/08-04-03.pdf

Regular expressions http://www.opengroup.org/onlinepubs/007908799/xbd/re.html

sed http://www.opengroup.org/onlinepubs/007908799/xcu/sed.html

YACC http://dinosaur.compilertools.net/yacc/

Collaborating Across Boundaries: Real-Time Geographically Distributed Software Development - FREE White Paper by MKS

Enabling software development teams that are geographically distributed to effectively collaborate on the same software projects can be challenging. Archive replication is not the only answer. Learn how to overcome global software project challenges and gain real-time access to project information, reporting and metrics. Learn more:

http://www.mks.com/mtj_collaborate_across_boundaries

You want to reach a specific software development community: **Software Testing & Quality, Agile, Rich Internet Application, Java or .NET** ? Check the new advertisement opportunities of Martinig & Associates on

http://www.martinig.ch/channel.php

Advertising for a new Web development tool? Looking to recruit software developers? Promoting a conference or a book? Organizing software development training? This classified section is waiting for you at the price of US $ 30 each line. Reach more than 50'000 web-savvy software developers and project managers worldwide with a classified advertisement in Methods & Tools. Without counting the 1000s that download the issue each month without being registered and the 60'000 visitors/month of our web sites! To advertise in this section, to place a page ad simply or a banner on our web sites network

http://www.methodsandtools.com/advertise.php

METHODS & TOOLS is published by **Martinig & Associates,** Rue des Marronniers 25,
CH-1800 Vevey, Switzerland Tel. +41 21 922 13 00 Fax +41 21 921 23 53 www.martinig.ch
Editor: Franco Martinig ISSN 1661-402X
Free subscription on : http://www.methodsandtools.com/forms/submt.php